Programmation python :

Cours 1: Programmation procedurale en python

Enseignant :

ATTA A. Ferdinand

28 juin 2021

Plan

choix d'un langage

- Grand nombre de langages existants :
  - avantages et inconvenients pour chacun
- Référence absolue : le couple C/C++
- Python prend de plus en plus de galons :
  - Python en tête du classement IEEE des 10 meilleurs langages en 2017 et 2018

Pourquoi python
caracteristiques d'implantations 1/6

- Portable sur :
  - UNIX et toutes ses variantes
  - WINDOWS et toutes ses variantes
  - MAC OS
  - etc...
- distribuer sous une licence open-source :
  - utiliser gratuitement, même pour construire des logiciels commerciaux payants.
- Extensible :
  - Possibilité d'interfacage avec le C

Pourquoi python
Python : dynamiquement typé (2/6)

- Java, C#, C++, etc.

```
1 int x = 1;
2 //x vaut 0 et ne peut jamais valoir 0.5
3 x = x/2;
```

- Python

```
1 x = 1;
2 x = x/2; # x egal à 0.5
```

Pourquoi python
Python : syntaxe simple (3/6)

python : cool ! ! !

Certains langages de programmation vous tueront avec des parenthèses, des accolades, des virgules et des deux points.

Avec python, vous passez moins de temps à déboguer la syntaxe et plus de temps à programmer : votre objectif.

Pourquoi python
Pourquoi faire long quand on peut faire court : "One-liners" 4/6

Solution élégante d'une ligne qui prend tout un bloc de code dans d'autres langages.
Par exemple : échanger x avec y

- Java, C#, C++, etc.

```
1 int temp  = x;
2 x = y;
3 y = temp;
```

- Python

```
1 x,y = y,x; # C'est tout !!!
```

Pourquoi python
des structures données intuitives 5/6

- Lists, tuples, dictionnaries, sets, etc...
- puissantes, encore très simple et intuitif à utiliser
- flexible : on peut les mixer à volonté
  - list de list
  - list de tuples
  - list de dictionnary
  - dictionnary de list
  - etc...

Pourquoi python
En vrai Africain (6/6)

Proverbe Africain

"DANS UN VILLAGE, QUAND TOUT LE MONDE MARCHE SUR LA TETE MON FRERE, MA SOEUR FAUT MARCHER SUR TA TETE AUSSI... "

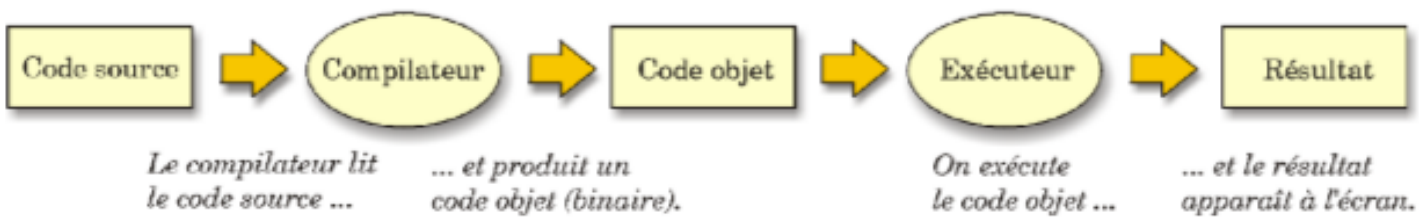
champs d'applications
Python : langage Généraliste

- Automatisez des tâches ennuyeuses(Administration système, réseau) : Langage de script
- Data Science(analyse de données, Machine Learning, etc...)
- Programmation scientifique
- Développement d'applications graphiques
- Les jeux vidéos :
- Développement d'applications application web : Django, Flask,etc.
- etc...

Langage machine et langage de programmation (1/2)

- Ordinateur = machine effectuant des opérations simples
- Opération = sequences de signaux electriques obeissant à la loi du tout ou rien : le binaire(0 ou 1)
- Toute information et programme doit être aussi traducte en une succession de 0 et 1 : traducteur ou interpreteur
  - Interprétation

  - Compilation



Langage machine et langage de programmation (2/2)
Et Python ?

- interpréteur permettant de tester n'importe quel petit bout de code
- compilation transparentes

Installation de python

- Python officiel :
  - https://www.python.org/downloads/
- Anaconda :
  - https://www.anaconda.com/download/
- Pycharm community :
  - https://www.jetbrains.com/pycharm/download/

Syntaxe et règles de nomenclature

- *nom_variable = valeur_initiale*
- suites de caractères alphanumériques et quelques caractères spéciaux
- mots réservés du langage interdit comme : for, print, while, assert, if.

```
1 #Initialisation d'une variable
2 #ayant pour nom 'ma_variable'
3 #et comme valeur 5.
4 ma_variable = 5
5 1e_variable = 5  #mauvaise nomenclature
```

Les différents types de variables

```
1  # Variable de type 'string' (chaine de caractères)
2  mon_nom = "ATTA"
3  # Variable de type 'integer' (nombre entier)
4  mon_nombre = 27
5  # Variable de type 'float' (nombre à virgule)
6  mon_nombre_a_virgule = 15.661415
7  # Variable de type 'boolean' (booléen en français)
8  # Un booléen peut  soit vrai, soit faux.
9  boolean_vrai = True
10 boolean_faux = False
```

Quelques manipulations de string et E/0 Standard

```
1  # Variable de type 'string' (chaine de caractères)
2  mon_prenom = "Ferdinand"
3  longueur = len(mon_prenom) #longueur de string
4  # iemecaractere
5  ieme_caractere = mon_prenom[0] #premier caractère
6  sous_chaine = mon_prenom[0:5] #sous chaine
7  #remplacement des n par b
8  nouvelle_chaine = mon_prenom.replace("n","b")
9  #concatenation avec +
10 conc_chaine = mon_prenom+nouvelle_chaine
11 longueur_string = str(longueur)#convertir en chaine
12 #pour le reste help(str)
13 print(mon_prenom)  #Sortie  standard
14 age = int(input("Entrez votre age")) #Entrée standard
```

Arithmétiques

```
1  add = 10 + 5 # Addition
2  soust = 10 - 5 # Soustraction
3  divise = 10 / 5 # Division
4  multipli = 10 * 5  # Multiplication
5  print(add, soust, divise, multipli)
6  puissance = 10 ** 5   # Puissance
7  # Modulo : Retourne le reste de la division
8  modulo = 10 % 5
9  # Division 'Floor'
10 # Arrondi le resultat de la division
11 # meme si les deux nombre sont des float
12 division_normale = 11.0 / 5.0 #le resultat est 2.2
13 division_floor = 11.0 // 5.0 #le resultat est  2
```

Logiques

```
1  #True si au moins le des propositions est vraie
2  a = True or False     # Retourne True
3  b = False or True     # Retourne True
4  c = True or True      # Retourne True
5  d = False or False    # Retourne False
6  # Renvoie True si les 2 propositions sont vraies
7  #False si au moins une des propositions est fausse.
8  a = True and False    # Retourne False
9  b = False and True    # Retourne False
10 c = False and False   # Retourne False
11 d = True and True     # Retourne True
12 # Retourne l'inverse de la proposition
13 a = not True          # Retourne False
14 b = not False         # Retourne True
```

Comparaisons

```
1  #  verifier si 2 propositions sont égales: ==
2  print(10 + 5 == 15)       # Retourne True
3  print(10 + 5 == 20)       # Retourne False
4  #  verifier si 2 propositions diffèrent : !=
5  print(10 + 5 != 15)       # Retourne False
6  print(10 + 5 != 20)       # Retourne True
7  # verifier si 1 élément est plus petit ou plus
8  #grand qu'1 autre : < et  >
9  print(5 < 10)   # Retourne True
10 print(10 < 10) # Retourne False
11 # verifier si 1 élément est plus grand ou égal
12 # ou plus petit ou égal a 1 autre  >= et <=
13 print(10 >= 10)# Retourne True
```

Assignations

définition

Il s'agit de la combinaison des opérateurs arithmétiques et celui d'affectation.

les combinaisons possibles

+=, **=, -=, *=, /=, %=

```
1 age = 15
2 age += 15
3 print(age) #affiche 30
```

Si .. sinon : if ... else

```
1 # Permet d'exécuter du code si condition sinon
2 # aucune condition n'est vraie
3
4 if 10 + 5 == 20:
5     print("10 + 5 n'est pas égal a 20")
6 else:
7     print('Il fallait écouter en cours de Maths!')
```

Si .. sinonsi ... else... : if ... elif ... else

```
1  # Permet de vérifier plusieurs
2  # conditions à la suite
3  a = 5
4  b = 10
5  if a < b:
6      print('a est plus petit que b')
7  elif a > b:
8      print('a est plus grand que b')
9  else:
10     print("a est donc forcement égal a b")
```

Boucle Pour : for

```
1  # La boucle For permet d'itérer à travers
2  # differents types de variables : chaine, list,
3  # dictionnary, etc...
4  # Cette boucle For affichera
5  # chaque lettre du mot bonjour
6  for lettre in 'Bonjour':
7      print(lettre)
8  # Cette boucle For affichera chaque nombre
9  # de 0 à 9
10 for i in range(0,10):
11     print(i)
```

Boucle Tant que : while

```
1  # La boucle While execute un bloc de code tant que
2  #la condition est vérifiée
3  # Cette boucle While affichera les nombres de 0-9
4  i = 0
5  while i < 10:
6      print(i)
7      i += 1
8  # ATTENTION! Une boucle While est dangereuse
9  # Si on ne modifie pas la condition, on
10 #créé une boucle infinie.
11 while True:
12     print('Cette boucle ne se termine jamais.')
13 # Il faut toujours permettre à la condition de
14 #changer afin de pouvoir sortir de la boucle.
```

List(1/2)

caractéristiques

- séquences d'éléments quelconques ordonnées
- mutable
- syntaxe : l = [< suite_elements_separés_par_virgule >]

Opérateurs et fonctions

- Accès à un élément : l[2], l[-2]
- Accès à une série d'éléments (slice) : l[2 :5], l[:-2]
- Longueur : len(l)
- Trier : l.sort() ou sorted(l)
- Renverser : l.sort(reverse=True), sorted(l,reverse=True)
- retirer un élément : l.remove(<element>), del l[<indice>]
- Rajouter en fin : l.append(<element>)
- Fusionner : l1.extend(l2)

List(2/2)

```
liste = ['Pierre', 'Paul']
# Rajoute l'élément à la fin de la liste
liste.append('Jacques')
# Insérer 1 élément à 1e position définie dans la liste
liste.insert(1, 'Kevin')
# Fusionne la liste ajoutée à la liste existante
liste.extend(['Bertrand', 'John','Pierre'])
#nombre d'occurences d'un éléments
pierres = liste.count("Pierre")
#parcourir une liste
for i in liste:
    print(i)
#comprehenson de liste :
autre_liste1 = [1, 2, 3, 4, 5]
autre_liste2 = [i*2 for i in autre_liste1]
autre_liste3 = [i*2 for i in autre_liste1 if i >= 3]
autre_liste4 = [i**2  if i >= 3 else i*2 for i in autre_liste1]
```

Tuple(1/2)

caractéristiques

- séquences d'éléments quelconques ordonnées
- immutable
- syntaxe : t = (< suite_elements_separés_par_virgule >)

Opérateurs et fonctions

- Accès à un élément : t[2], t[-2]
- Accès à une série d'éléments (slice) : t[2 :5], t[:-2]
- Longueur : len(t)
- Appartenance : in
- Concaténation : +
- Duplication : *
- maximum/minimum : max/min

Les 4 derniers points sont applicables également aux listes et au string

Tuple(2/2)

```
# Création d'un tuple
mon_tuple = (1, 2, 3)
# Recupération de la valeur minimale
valeur_min = min(mon_tuple)
# Recupération de la valeur maximale
valeur_max = max(mon_tuple)
#duplication par facteur = 2
autre_tuple = mon_tuple*2
```

Dictionnary(1/2)

caractéristiques

- Table de hachage
- clés de type immutable et objets de type mutable ou pas
- syntaxe : h1={} ou h1={"*key*1" : *value*1,"*key*2" : *value*2}

opérations et fonctions

- accéder à élément : h1["cle"]
- les clés : keys()
- les valeurs : values()
- les éléments : items()
- vérifier si clé : has_key(cle_candidat)
- effacer le dictionnaire : clear()

Dictionnary(2/2)

```
dico1 = {'Pierre': 40, 'Paul': 25}# Créer d'un dictionnaire
print(dico1['Pierre'])#Acceder a element
print(dico1.keys())
print(dico1.values())
dico1.items()#liste de tuples(cle,valeur)
dico1['Paul'] = 30 #Mis à jour
dico1.update({'Pierre': 42})
for cle in dico1.keys():  #les cles par une boucle
    print(cle)
for valeur in dico1.values():
    print(valeur) #les valeurs par une boucle
for cle,valeur in dico1.items():
    print (cle, valeur) #les cles et valeurs par une boucle
del dico1['Pierre'] # Supprimer avec del
dico2 = {'Pierre': {'age': 40, 'profession': 'banquier'}, \
'Paul': {'age': 25, 'profession': 'ingenieur'}}
print(dico2['Pierre']['age'])
```

Fonction 1/5

```
# Syntaxe
def nom_de_la_fonction(<parametres_formels_eventuels):
    code_fonction
# Appel de la fonction
nom_de_la_fonction(<parametres_effectifs_eventuels)
# Fonction sans paramètre et renvoie rien
def affiche_paragraphe():
    print('Bonsoir, Ceci est un bloc de code')
    print("qui s'affiche quand la fonction est appelée")
#Fonction avec paramètre et sans valeur de retour
def addition(a=5, b=10):
    print(a + b)
# Fonction avec paramètre et avec valeur de retour
def addition(a=5, b=10):
    c = a + b
    return c
```

Fonction 2/5 : arguments variables

- On passe les paramètres depuis ou vers des conteneurs
- l'opérateur * convertit des arguments non nommés en tuple
- l'opérateur ** convertit des arguments nommés en dictionnaire

```
1  # Utilisation de l'opérateur *
2  def test_var_args(farg, *args):
3      print('formel arg:', farg)
4      for arg in args :
5          print('autre arg:', arg)
6  test_var_args(1,'two', 3)
7  # Utilisation de l'opérateur **
8  def test_var_kwargs(farg, **kwargs):
9      print ("formel arg:", farg)
10     for key in kwargs :
11         print('autre clé arg: {}:{}'.format(key,kwargs[key]))
12 test_var_kwargs(farg=1,myarg2='two', myarg3=3)
```

Fonction 3/5 : Fonctions anonymes

- Utilisation du mot clé **lambda** au lieu de **def**
- Par exemple :

```
1 #creation d'une fonction utilisation
2 #expression lambda
3 dernier = lambda t: t[-1]
4 #ou fonction classique avec def
5 def lastElement(t):
6     return t[-1]
7 #utilisation de fonction anonyme
8 maliste = [8,9,0,5]
9 print(dernier(maliste))
```

Fonction 4/5 : Fonctions imbriquées

- Une instruction **def** dans le corps d'une fonction → fonction imbriquée
- Par exemple :

```
1  def pourcent(a,b,c):#Exemple1
2      def pc(x, total = a+b+c):
3          return (x/total)*100.0
4      print("Les pourcentages sont: ", pc(a), pc(b), pc(c))
5  def faire_une_fonction():#Exemple2
6      fraichitude = "so fresh !"
7      def une_fonction_toute_fraiche():
8          return fraichitude
9      print(une_fonction_toute_fraiche())
10 faire_une_fonction()
```

Fonction 5/5 : closure

- Problématique : comment une fonction imbriquée peut-elle avoir accès aux données de la fonction qui l'englobe à l'extérieur de cette dernière ?
  - Closure = fonction avec un « contexte » de variables affectées

```
1  #accéder à la variable fraichitude de
2  # l'exemple 2(diapo précédente)
3  #à l'extérieur de  la fonction faire_une_fonction()?
4  def faire_une_fonction():#  closure
5      fraichitude = "so fresh !"
6      def une_fonction_toute_fraiche():
7          return fraichitude
8      return une_fonction_toute_fraiche # Fin closure
9  fonction = faire_une_fonction()
10 print(fonction())
```

Module et package(1/3)

Définitions

Un module est un fichier (*.py) contenant la définition d'un ensemble de fonctions, variables, constantes, classes, etc.
Un package est un répertoire contenant des modules (et éventuellement des packages). Il contient un fichier spécial forcément nommé __init__.py
Plusieurs manières d'utiliser un module

- import nom_module as nom_alias
- from nom_module import nom_fonction
- from nom_module import *
- from nom_package.nom_module import *
- etc...

Module et package(2/3)

- Module : fichier contenant de la definition de fonctions, variables, classes, constantes
- Exemple de module : puissance.py

```
1 def carre(valeur):
2     resultat = valeur**2
3     return resultat
4 def cube(valeur):
5     resultat = valeur**3
6     return resultat
```

Module et package(3/3)

- Package = Ensemble de modules

```
Sound/                          Paquetage de niveau supérieur
      __init__.py               Initialisation du paquetage sons
      Formats/                  Sous-paquetage pour la conversion des formats de fichiers
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Sous-paquetage pour les effets sonores
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Sous-paquetage pour les filtres
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Résumé : structure d'un programme Python type

```
# -*- coding:Utf8 -*-

#####################################
# Programme Python type             #
# auteur : G.Swinnen, Liège, 2009   #
# licence : GPL                     #
#####################################


#####################################
# Importation de fonctions externes :

from math import sqrt


#################################
# Définition locale de fonctions :

def occurrences(car, ch):
    "Cette fonction renvoie le \
     nombre de caractères <car> \
     contenus dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


###############################
# Corps principal du programme :

print("Veuillez entrer un nombre :")
nbr = eval(input())

print("Veuillez entrer une phrase :")
phr = input()
print("Entrez le caractère à compter :")
cch = input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print("La racine carrée du cube", end=' ')
print("du nombre fourni vaut", end=' ')
print(rc)
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- *Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales).*
- *Les définitions locales de fonctions et/ou de classes.*
- *Le corps principal du programme.*

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes.
Vous pouvez vous-même définir de tels modules.

La définition d'une fonction comporte souvent une liste de PARAMÈTRES.
Ce sont toujours des VARIABLES, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit toujours inclure au moins quatre éléments :
- *l'initialisation d'une variable 'compteur' ;*
- *l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;*
- *le bloc d'instructions à répéter ;*
- *une instruction d'incrémentation du compteur.*

La fonction "renvoie" toujours une valeur bien déterminée au programme appelant.
Si l'instruction 'return' n'est pas utilisée, ou si elle est utilisée sans argument, la fonction renvoie un objet vide : 'None'.

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'ARGUMENTS, lesquels peuvent être des valeurs, des variables, ou même des expressions.

Ouverture de fichier

```
1  #Emplacement propre à ma machine donc le changer
2  #selon votre machine
3  chemin = '/home/atta/file.txt'
4  # Ouvrir un fichier en mode 'Read' (Lecture)
5  #le fichier file.txt doit exister effectivement
6  #sur votre disque
7  f = open(chemin, 'r')
8  # Ouvrir un fichier en mode 'Write' (écriture)
9  #le fichier file.txt peut ne pas exister
10 #effectivement
11 #sur votre disque, une création du fichier au cas
12 #écheant sera effectuée
13 f = open(chemin, 'w')
14 # Ouvrir un fichier en mode 'Append' (Ajout)
15 f = open(chemin, 'a')
```

Fermeture de fichier ou construction *with*

```
1 #fermeture de fichier
2 f.close()
3 #Construction with est sure car le fichier est
4 #automatiquement fermé après l'exécution
5 #des instructions du bloc with.
6 with open('/home/atta/file.txt',mode='r',encoding='utf8') as f:
7     pass
8 print(f.closed)
```

Écriture dans un fichier

```
1  chemin = '/home/atta/file.txt'
2  #Methode1 : sans with
3  #Pour écrire dans un fichier et rajouter du contenu
4  # à sa suite
5  f = open(chemin, 'a')
6  f.write('Contenu ajouté')
7  f.close()
8  #Methode2 : avec  with
9  #mode="w" écrase le contenu exisant
10 with open(chemin, mode='w', encoding='utf8') as f:)
11     f.write('Contenu1 ajouté')
12     f.write('Contenu2 ajouté')
```

Lecture de fichier

```
1  chemin = '/home/atta/file.txt'
2  f = open(chemin, 'r')
3  # Pour récupérer le contenu complet du fichier dans
4  #une chaine de caractere
5  f.read()
6  f.readlines()# Pour recuperer une liste de lignes
7  # Pour boucler à travers chaque ligne
8  # Avec une boucle For
9  for line in f.readline():
10         print(line)
11 line = f.readline()
12 while line: # Avec une boucle While
13     print(line)
14         line = f.readline()
15 #Possibilité d'utiliser with comme pour écriture
16 with open(chemin, mode='r', encoding='utf8') as f:)
17         for l in f: print(l)
```

Vérifier les propriétés d'un fichier

```
1  chemin = '/home/atta/file.txt'
2  f = open(chemin, 'r')
3  # Pour vérifier si un fichier est ferme
4  f.closed
5  >>> False
6  # Pour vérifier le mode dans lequel le fichier a été ouvert
7  f.mode
8  >>> 'r'
9  # Pour verifier le chemin du fichier
10 f.name
11 >>> '/home/atta/file.txt'
```

- Pour allez plus loin :
  - Les expressions génératrices
  - fonctions génératrices
  - les tuples nommés
  - etc .
- Place au TP ! ! !

References

1-Gerard Swinnen : Apprendre a programmer avec Python.pdf
2-https://www.developpez.com/actu/217533/Meilleurs-langages-en-2018-selon-l-IEEE-Python-conforte-sa-place-de-leader-grace-a-son-ascension-dans-le-machine-learning-et-l-embarque/
3-http://www.linux-center.org/articles/9812/python.html
4-Mark Lutz : Python précis et concis
5-http://sametmax.com/closure-en-python-et-javascript/