Rem: sortie = entrée => dans programme info après déf cad réponse = données & calcul

LINGE1225 : Programmation en économie et gestion

Cours 2

Fonctions originales et exécution conditionnelle

François Fouss & Marco Saerens

Année académique 2020-2021

Livre de référence

- Chapitre 3 : Contrôle du flux d'exécution
- Chapitre 7 : Fonctions originales

2

Plan

1. Opérateurs
2. Exécutions conditionnelles
3. Fonctions originales
4. Fonctions prédéfinies (rappel)
   - Bibliothèques de fonctions prédéfinies
   - Fonctions utiles
5. Application

Chemin et flux

- Le « chemin » suivi par Python à travers un programme est appelé un flux d'exécution, et les constructions qui le modifient sont appelées des instructions de contrôle de flux.
- Ordre d'exécution par défaut :
  - Séquence: Exécution des instructions les unes à la suite des autres, dans l'ordre d'écriture

4

Contrôle de flux

- Certaines déclarations de programmation modifient cet ordre, permettant de :
  - décider s'il faut exécuter ou pas une déclaration particulière: on parle de condition (sélection)
  - ou effectuer une déclaration encore et encore, de manière répétitive: on parle de répétition (cf. cours 3)
- Ces décisions sont basées sur une expression booléenne (aussi appelé condition) évaluée à tout moment à vrai ou faux

5

Opérateurs de comparaison

Opérateurs logiques

- Les expressions booléennes utilisent des opérateurs logiques :

NOT	Logical NOT
AND	Logical AND
OR	Logical OR

- Ils produisent tous des résultats booléens
- Logical NOT est un opérateur unaire
- Logical AND et Logical OR sont des opérateurs binaires

7

Opérateurs logiques

- Négation logique (**not**)

En Python, la négation est représenté par le not

a	not a
True	not True = False
False	not False = True

- ET logique (**and**)

a/b	True	False
True	True and True = True	True and False = False
False	False and True = False	False and False = False

- OU logique (**or**)

a/b	True	False
True	True or True = True	True or False = True
False	False or True = True	False or False = False

8

Opérateurs de comparaison (résultat booléen)

- Une condition utilise souvent une des opérations d'égalité ou les opérateurs relationnels de Python, qui retournent tous des résultats booléens:

==	égal à
!=	pas égal à
<	plus petit que
>	plus grand que
<=	plus petit ou égal que
>=	plus grand ou égal que

Remarque : différence d'écriture entre l'opérateur d'égalité (==) et l'opérateur d'assignation (=)

Assigne une valeur à une variable

Compare 2 valeurs et renvoi une valeur booléen (true/false)

9

Opérateurs de comparaison (booléens)

- Opérandes quelconques donneront un résultat booléen

- Egalité entre valeurs == : est possible entre toutes paires de valeurs.

- Comparaison de valeurs

!=, <,>, <=, >=

```
#égalité et comparaison

    #égalité

    5 == 3 + 2                  True
    5 == 6                      False
    5 == 5.0                    True
    5 == "5"                    False
    "toto" == "to" + "to"       True



    #comparaison

    4 <= 1                      False
    4 >= 2                      True
    4 != 2                      True
```

Car 5 = int et « 5 » = string

10

Exécution conditionnelle

Exécutions conditionnelles

- Une exécution conditionnelle permet de choisir la prochaine instruction qui sera exécutée
- Une exécution conditionnelle nous laisse la possibilité de prendre des décisions simples.
- Les exécutions conditionnelles de Python sont :
  - *if*
  - *if-else*

12

Exécution conditionnelle

➢ L'exécution `if` a la syntaxe suivante :

La valeur finale doit valoir True ou False

La `condition` doit être une expression booléenne. Cela doit être évaluée de façon vrai ou fausse.

`if` est un mot réservé à Python

= indentation

```
if condition :
    # déclaration
    #
    #
```

Si la `condition` est vraie, la `déclaration` est effectuée.

Si c'est faux, la `déclaration` est passée

13

Instruction conditionnelle

➢ Premièrement, la condition est évaluée. La valeur de x est soit plus grande ou plus petite que la valeur de y.

➢ Si la condition est vraie, la déclaration est effectuée

➢ Si la condition est fausse, la déclaration est passée

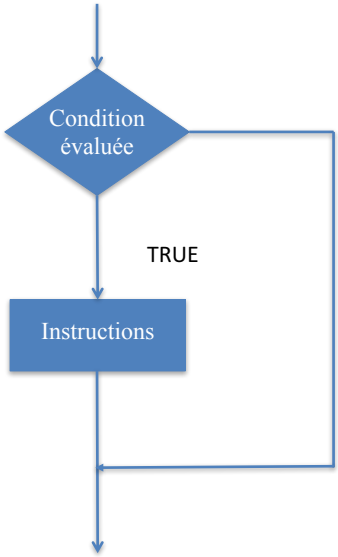
➢ Finalement, la fonction `print()` permet d'afficher la variable

```
#Déclaration conditionnelle

y = 5          #affectation
x = 2

if x<y:          #condition
    x = x + 1    #déclaration
print(x)
```

indentation

14

Logique




```
1  x = 2
2  if x > 6 :
3      x += 3
4      x = 5
5  print (x)
```

ex si cond est fausse

Résultat : x vaut 2

```
1  x = 10
2  if x > 6 :
3      x += 3
4      x += 2
5  print (x)
```

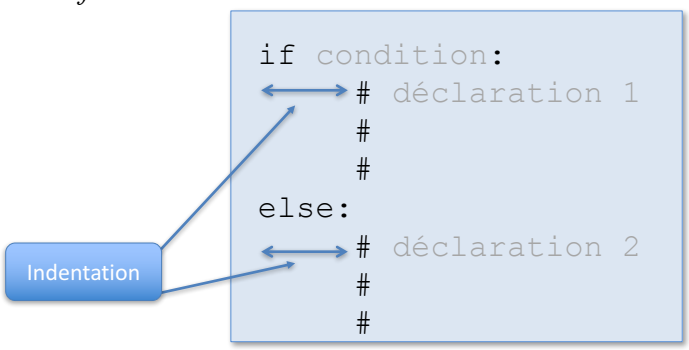
ex si cond est vraie

Résultat : x vaut 15

15

if-else

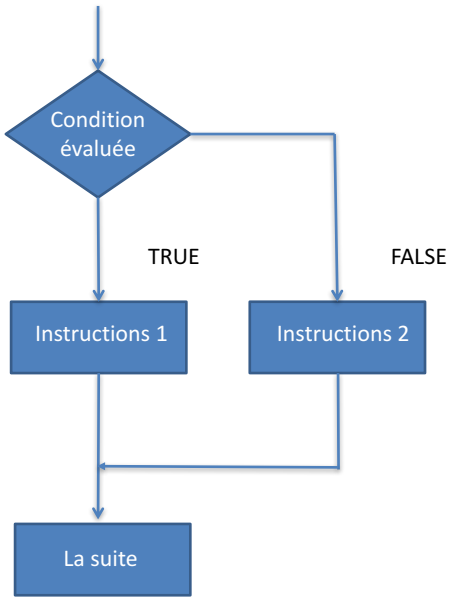
- Un *else clause* peut être ajouté à une exécution `if` pour former une *if-else* exécution



- Si la `condition` est vrai, la `déclaration 1` est effectuée mais pas la `déclaration 2`. Si la `condition` est fausse, la `déclaration 2` est effectuée mais pas la `déclaration 1`.
- Une des deux déclarations sera effectuée mais pas les deux.

16

if-else




```
#Instruction conditionnelle

if condition:
↔ #
    # Faire quelque chose
    #
else :
↔ #
    # Faire autre chose
    #
#
# La suite
#
```

```
x = 10
if x > 6 :
    x = 6
else :
    x = 0
print (x)
```

Résultat : x vaut 6

```
x = 2
if x > 6 :
    x = 6
else :
    x = 0
print (x)
```

Résultat : x vaut 0

17

Règle de syntaxe Python

- Les limites des instructions et des blocs sont définies par la mise en page
- Les espaces commentaires sont normalement ignorés

```
#Instruction conditionnelle

y = 5          #affectation
x = 2

if x<5:          #condition
↔ x = x + 1    #déclaration

print(x)
```

18

Instructions composées : blocs d'instructions

- Toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées (i.e., bloc d'instructions)
- S'il y a plusieurs instructions indentées, elles doivent l'être exactement au même niveau

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

19

Règle de syntaxe Python

- Instruction composée : en-tête, double point, bloc d'instructions indenté

Bloc d'instruction :

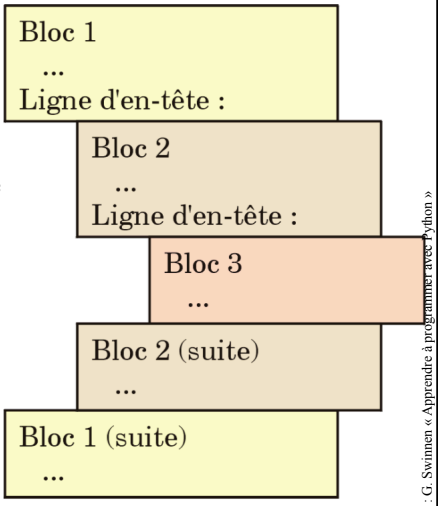
If, elif, while, def, ..

Les blocs sont délimités par l'indentation : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière
Remarque : bloc1 ne peut pas lui-même être écarté de la marge

Exemple :

```
1  x = 10
2  if x > 6 :
3      x -= 1
4      if x > 7 :
5          x -= 2
6          if x > 9 :
7              x = 0
8          y = 2
9      z = 1
10 print (x,y,z)
```

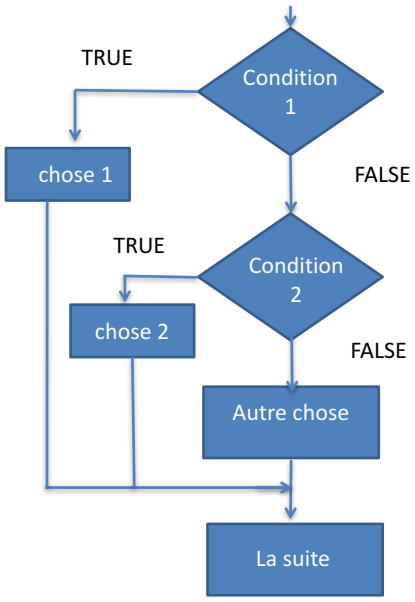
4 blocs



20

Source : G. Swinnen « Apprendre à programmer avec Python »

Conditionnelle en chaîne




```
#Instruction conditionnelle

if condition1:
    #
    # Faire quelque chose 1
    #
elif condition2:
    #
    # Faire autre chose 2
    #
else:
    #
    #
    #

#
# La suite
#
```

contraction de 'else if', même fonctionnement que 'else'.

21

Elif = sinon

Le « elif » est comme un « else » mais avec une condition.

!!! Il peut y avoir autant d'elif d'affilé que l'on veut

```
x = 1
if x == 2 :
    x = 1
elif x == 1 :
    x = 0
elif x == 0 :
    x = -10
else :
    x = 5
print (x)
```

Résultat : x vaut 0

```
x = 1
if x == 2 :
    x = 1
if x == 1 :
    x = 0
if x == 0 :
    x = -10
else :
    x = 5
print (x)
```

Résultat : x vaut -10

La différence entre le code de gauche et celui de droite est qu'un « elif » est exécuté seulement si aucune instruction conditionnelle précédente n' été évaluée à « True ». Tandis qu'un « if » ne dépend pas des instructions conditionnelles précédentes.

22

Exécutions conditionnelles - Instructions imbriquées

- Plusieurs instructions composées peuvent être imbriquées les unes dans les autres

```
if embranchement == "vertébrés":                    # 1
    if classe == "mammifères":                      # 2
        if ordre == "carnivores":                   # 3
            if famille == "félins":                 # 4
                print("c'est peut-être un chat")    # 5
        print("c'est en tous cas un mammifère")     # 6
    elif classe == "oiseaux":                       # 7
        print("c'est peut-être un canari")          # 8
print("la classification des animaux est complexe") # 9
```

- Importance des sauts à la ligne et indentation

23

Fonctions originales

24

Fonction

- Fonction prédéfinie
  - Fonction dont l'utilité est connue et régulièrement demandée
  - Fonction dont le code a déjà été écrit (vous pouvez les utiliser sans les coder)
  - Fonction qui fait partie de la trousse à outils existante
  - Fonction qui peut être appelée à tout moment
- Fonction originale
  - Fonction qui n'existe pas encore
  - Fonction qui revêt d'une utilité spécifique ou neuve pour le programmeur
  - Fonction dont le code doit être écrit
  - Fonction qui fera ensuite (i.e., après l'écriture du code) partie de la trousse à outils du programmeur
  - Fonction qui pourra ensuite (i.e., après l'écriture du code) être appelée à tout moment

25

Définir une fonction

Pourquoi une fonction ?

- Pour décomposer un problème en plusieurs sous problèmes plus simples qui seront étudiés séparément
- Lorsqu'une même séquence d'instructions doit être utilisée à plusieurs reprises dans un programme

La syntaxe Python pour la définition d'une fonction est la suivante :

/!\ Indentation

```
def nomDeLaFonction(liste de paramètres):
    …
    bloc d'instructions
    …
```

26

Fonction simple sans paramètre

La fonction ne prend pas d'argument

Nom de la fonction

Indentation

Appel de fonction

```
#Fonction simple sans paramètre

def table7():
    n=1
    while n<11:
        print(n * 7, end = ' ')
        n = n + 1
table7()
```

27

Fonction simple avec paramètres

Faisons une fonction mais pour la table de 9

→Il faut réécrire une fonction

Ne serait-il pas possible de définir une fonction qui soit capable d'afficher n'importe quelle table à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cela sera notre **argument** de la fonction. Pour la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis = **paramètre**

28

Fonction simple avec paramètres

Fonction avec paramètre

paramètre

paramètre

Appel de fonction avec argument = 13

Appel de fonction avec argument = 21

```
#Fonction simple avec paramètre

def tab(base):
    n = 1
    if n <11:
        print(n * base, end=" ")
        n = n +1

tab(13)

tab(21)
```

29

Fonction simple avec plusieurs paramètres

```
#Fonction avec plusieurs paramètres

def tableMulti(base, debut, fin):
    print("Fragment de la table de multiplication par", base, ":")
    n = debut
    while n <= fin :
        print(n, "x", base, "=", n * base)
        n = n +1

tableMulti(8, 13, 17)
```

Résultat du programme :

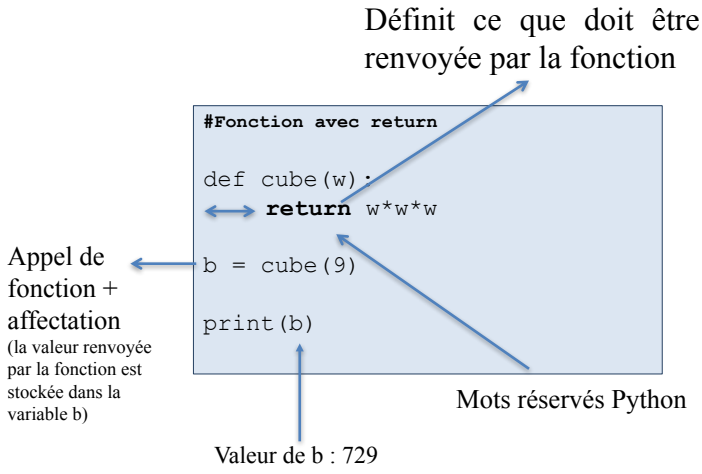
```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Remarque :

- L'insertion de plusieurs paramètres se fait en rajoutant les paramètres entre les parenthèses qui suivent la fonction, en les séparant d'une virgule
- Lors de l'appel de la fonction, les arguments utilisés doivent être fournis dans le même ordre que celui des paramètres correspondants

30

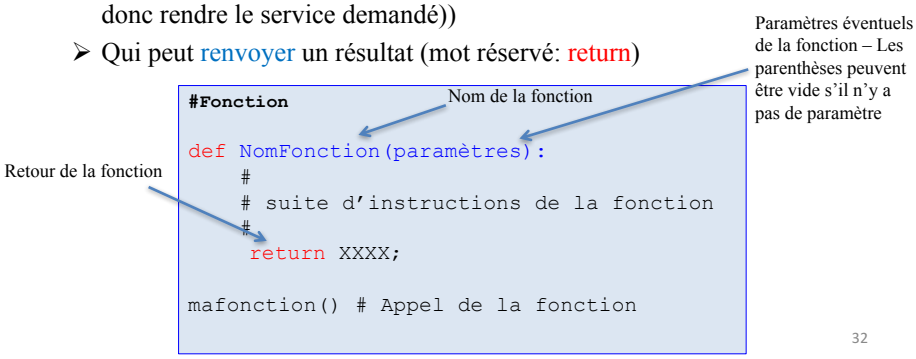
Fonction simple avec return



31

Fonction: Résumé

- Une fonction = Suite d'instructions que l'on peut appeler avec un nom
- Mot réservé def pour indiquer que l'on définit une fonction
  - Qui doit posséder un nom unique
  - Qui peut posséder un ou plusieurs paramètres (= des données à fournir comme input à la fonction afin qu'elle puisse s'exécuter correctement (et donc rendre le service demandé))
  - Qui peut renvoyer un résultat (mot réservé: return)



32

Attention

- « Appeler » une fonction = demander l'exécution de la suite d'instructions de la fonction
- ATTENTION à bien différencier
  - La création (du code) de la fonction
  - L'appel (de l'exécution du code) de la fonction

```
#Création fonction

   def mon_age():       #définition de la fonction
       return 26;        #code de la fonction
```

```
#Appel fonction

   mon_age()       #appel de la fonction
    26            #exécution du code de la fonction
```

- Exemple dia suivante

33

Fonction - Exemple

La fonction ne prend pas d'argument

```
1  def mon_age():
2      return 26
3
4  x = mon_age()
5  print(x)
```

Valeur que « return » (renvoie) la fonction

Résultat : x vaut 26

La fonction ici a comme nom : mon_age et elle ne prend pas d'argument.

mon_age() définit un ensemble d'instructions qui doit être exécuté quand on l'appelle (en l'occurrence, elle n'en a qu'une : « return 26 »)

L'instruction « return » renvoie une valeur, c'est-à-dire : c'est le résultat de la fonction. Cette valeur peut être stockée dans une variable (ici : elle est stockée dans la variable x)

34

RAPPEL

Fonctions prédéfinies

Bibliothèques de fonctions prédéfinies

Importer un module de fonctions

RAPPEL

Nous avons vu des fonctions intégrées au langage Python telles que input(), len(),…

Cependant les fonctions intégrées au language sont relativement peu nombreuses. Les autres sont regroupées dans des fichiers séparées que l'on appelle des **modules.**

- **Modules** = fichiers qui regroupent des ensembles de fonctions prédéfinies.
- **Bibliothèques** = module qui regroupe des ensembles de fonctions apparentées

Syntaxe :

```
from <module> import <fonction>
```

Importer un module de fonctions

RAPPEL

Exemple :

Le module math, contient des définitions de nombreuses fonctions mathématiques telles que sinus, cosinus, tangente, racine carrée etc.

Importer toutes les fonctions

```
from math import *
```

nom du module

- Interprétation : Il faut inclure dans le programme courant toutes les fonctions signe * du module `math`, lequel contient une bibliothèque de fonctions mathématiques préprogrammées.

37

Module math

RAPPEL

Sans import le module math, le programme nous renverrait une erreur disant qu'il ne connait pas « sqrt() »

```
#Utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6

print('racine carrée de', nombre, '=', sqrt(nombre))
print('sinus de', angle, 'radians', '=', sin(angle))
```

Utilisation de fonction du module math

Utilisation de fonction du module math

Ce qui donne :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

38

Caractéristiques des fonctions

RAPPEL

- Une fonction apparaît sous la forme d'un nom quelconque associé à des parenthèses
  exemple : `sqrt()`
- Dans les parenthèses, on transmet à la fonction un ou plusieurs arguments
  exemple *:* `sqrt(121)`
- La fonction fournit une valeur de retour (on dira aussi qu'elle « retourne », ou mieux, qu'elle « renvoie » une valeur)
  exemple *:* 11.0

39

Conseils

RAPPEL

- <math> est une toute petite partie des modules de Python.
- N'hésitez pas à regarder dans la documentation bibliothèque de Python les très nombreuses différentes fonctions disponibles.
- ! Vous ne pourrez utiliser que certains modules pour l'examen

40

Random

RAPPEL

- Programmes déterministes = programmes qui feront toujours la même chose chaque fois qu'on les exécute.
- Dans son module `random`, Python propose toute une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques.

Exemple

```
from random import *
```

41

Random

RAPPEL

« n » est la valeur de l'argument passé à la fonction. Cette valeur est déterminée lors de l'appel à la fonction.

```
def list_aleat(n):
    s = [0]*n
    for i in range(n):
        s[i]= random()
    return s

list_aleat(3)
[0.3322011398880431, 0.5642475883164183, 0.591966613411697]

list_aleat(3)
[0.941262175783142, 0.16765659221691698, 0.9420901408904373]
```

La fonction `random` permet de créer une liste de nombres réels aléatoires, de valeur comprise entre zéro et un

- Construction d'une liste de zéros de taille n et ensuite remplacement des zéros par des nombres aléatoires

42

Exercice

RAPPEL

- Créez un programme qui simule le lancé de deux dés, faites la somme et affichez le résultat de cette somme

43

Solution

RAPPEL

On spécifie au programme d'où vient « randint ». Pour ce faire, on fait : `<module>.<fonction()>` où le module a été import préalablement.

```
import random

des1 = random.randint(1,6)

des2 = random.randint(1,6)

résultat = des1 + des2

print(résultat)
```

44

RAPPEL

Fonctions prédéfinies

Fonctions utiles

45

input()

RAPPEL

`input()` fonction

- Permet d'interagir avec l'utilisateur
- L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <Enter>.
- Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a saisi.

```
#Fonction input()

prenom = input("Entrez votre
prénom : ")

print("Bonjour," prenom)

print("Veuillez entrer un
nombre positif quelconque : " ,
end="  ")

ch = input()
nn = int(ch) #conversion chaine
                en nbre entier

print("Le carré de ", nn,
"vaut", nn**2)
```

46

type()

RAPPEL

`type()` fonction

- Permet d'afficher le type de la variable entrée en paramètre

```
#Fonction type()

type(25)
<class 'int'>

type("bonjour")
<class 'str'>

type(24.2)
<class 'float'>

type(True)
<class 'bool'>
```

47

len()

RAPPEL

`len(parameter)` fonction

- Calcule le nombre d'éléments de l'argument `parameter` passé en paramètre
- Paramètre d'une fonction
  - Input nécessaire à l'exécution de la fonction
- Si parameter est une chaîne de caractères, `len(parameter)` renvoie le nombre de caractères de `parameter`

```
#Fonction len(parameter)

len("bonjour")
7

var1 = Coucou
len(var1)
6
```

48

print()

RAPPEL

`print()` fonction

- Permet d'afficher n'importe quel nombre de valeurs fournies en arguments
- Par défaut, ces valeurs seront séparées les unes des autres par un espace et se terminera avec un saut à la ligne.
- Possibilité de remplacer le séparateur par défaut par un caractère quelconque grâce à l'argument sep.

```
#Fonction print()

print("Bonjour", "à", "tous",
sep="*")

Bonjour*à*tous

print("Bonjour", "à", "tous",
sep="")

Bonjouràtous

print("Bonjour", "à", "tous")

Bonjour à tous
```

49

range()

RAPPEL

`range(end)`

- Renvoie une liste de nombres entiers allant de `0` à `end-1`

`range(start,end)`

- Renvoie une liste de nombres entiers allant de `start` à `end-1`

`range(start,end,incr)`

- Renvoie une liste de nombres entiers allant de `start` au plus grand nombre incrémenté inférieur à `end-1`, par incrément de `incr`

```
#Fonction range(end)

range(5)
Crée la liste [0,1,2,3,4]

#Fonction range(start,end)

range(0,10)
Crée la liste
[0,1,2,3,4,5,6,7,8,9]

#Fonction range(start,end,incr)

range(10,35,5)
Crée la liste [10,15,20,25,30]
```

50

abs()

RAPPEL

`abs(nbre)` fonction

- Renvoie la valeur absolue de l'argument `nbre` passé en paramètre
- Si l'argument est un nombre entier ou flottant, le résultat est un nombre entier ou flottant

```
#Fonction abs(nbre)

abs(-3)
3

var1 = -3.14
abs(var1)
3.14
```

51

pow() et sqrt()

RAPPEL

`pow(x,y)` fonction

- Renvoie la valeur de `x` élevée à la puissance `y`

`sqrt(x)` fonction

- Renvoie la racine carrée de `x`

```
#Fonction pow(x,y)

pow(2,3)
8

#Fonction sqrt(x)

sqrt(16)
4
```

52

Application

Maximum de 3 chiffres

Définissez une fonction **maximum(n1,n2,n3)** qui renvoie le plus grand de 3 nombres **n1, n2, n3** fournis en arguments. Par exemple, l'exécution de l'instruction : **print(maximum(2,5,4))** doit donner le résultat : **5.**
(exercice 7.3 du livre de référence (page 74))

54

Maximum de 3 chiffres

Solution :

```
def maximum(n1, n2, n3):
    "Renvoie le plus grand de trois nombres"
    if n1 >= n2 and n1 >= n3:
        return n1
    elif n2 >= n1 and n2 >= n3:
        return n2
    else:
        return n3
# test :
print(maximum(4.5, 5.7, 3.9))
print(maximum(8.2, 2.1, 6.7))
print(maximum(1.3, 4.8, 7.6))
```

55