e-PyLearn

Syllabus:

1 Introduction

2 Data Structure

3 Recursion

Course 1 - Introduction: Getting Started: Python and IDLE

This handout will cover how to set up Python and introduce you to IDLE, the Python development environment we will be using throughout this course.

Setting up PythonOn Linux

Python should be set up correctly on the Linux athena* machines. Type 'idle' at the command prompt to ensure that everything is working correctly. This should start up the Python development environment IDLE. On your own machineIf you are working on your own machine, you will probably need to install Python. We will be using the standard Python software, available here. You should download and install version 2.6.x, NOT 2.7.x or 3.x All MIT Course 6 classes currently use a version of Python 2.6. Windows: Go to the website and download the windows MSI installer for either x86 or x86-64, depending on which version of Windows you are running. Mac OS X: Download and install the Mac Installer disk image from the site. Other Linux: Check which version of Python you have by running python -V at a terminal. If you have a newer version of Python already installed - eg Python 2.7.x or 3.1.x, you can set Python 2.6 as the default by following the instructions listed here. Otherwise, you should be able to do one of the following options:

```
sudo apt-get install python2.6
```

if you don't already have Python 2.6 installed; if you do, run

```
sudo apt-get install idle
```

To install Idle for Python 2.6. If you have Python and Idle installed with a *newer* version of Python (eg Python 3.1... python 2.7 won't cause a conflict for 6.189 and you can leave it alone for now), you'll want to instead run these two commands to install the correct version of Idle:

```
sudo apt-get install idle-python2.6
```

```
sudo ln -s /usr/bin/idle-python2.6 /usr/bin/idle
```

You should then be able to run Idle by simply running idle& from the command prompt. If you would rather compile from source, visit the Python 2.6.4 release page for compressed tarballs. If you're having problems, please ask an LA for assistance. Warning: On the Python homepage, the latest version available for download is actually 3.0. Do not install this! This version is not backwards compatible with the code that you'll be writing in this course (for

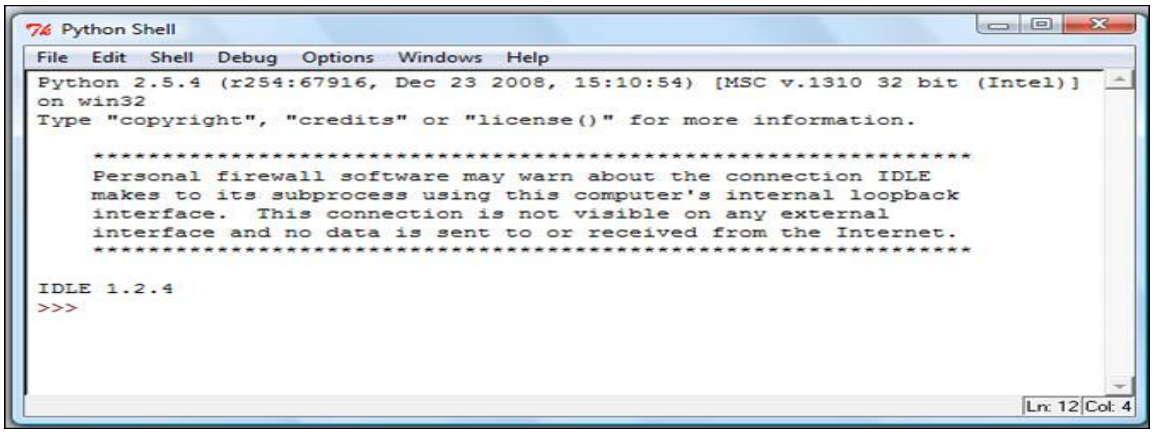
example, you have to type print("test") instead of print "test"). Instead, be sure to download the version listed above.

Using IDLE

IDLE is the standard Python development environment Its name is an acronym of "**I**ntegrated **D**eve**l**opment environment". It works well on both UNIX and Windows platforms. It has a Python shell window, which gives you access to the Python interactive mode. It also has a file editor that lets you create and edit existing Python source files. During the following discussion of IDLE's features, instead of passively reading along, you should start IDLE and try to replicate the screenshots.

Interactive Python shell

When you start up IDLE, a window with an interactive Python shell will pop up:



You can type Python code directly into this shell, at the '>>>' prompt. Whenever you enter a complete code fragment, it will be executed. For instance, typing:

You can type Python code directly into this shell, at the '>>>' prompt. Whenever you enter a complete code fragment, it will be executed. For instance, typing:

```
>>> print "hello world"
```

and pressing ENTER, will cause the following to be displayed:

```
hello world
```

Try typing an underscore (_). Can you see it? On some operating systems, the bottoms of hanging letters such as 'g' or 'y', as well as underscorces, cannot be seen in IDLE. If this is the case for you, go to Options -> Configure IDLE, and change the size of the default font to 9 or 11. This will fix the problem! IDLE can also be used as a calculator:

```
>>> 4+4 8>>> 8**3 512
```

Addition (+), subtraction (−), multiplication (*), division (/), modulo (%) and power (**) operators are built into the Python language. This means you can use them right away. If you want to use a square root in your calculation, you

can either raise something to the power of 0.5 or you can *import* the *math* module. Do not worry about what it means right now, we will cover this later during the course. Below are two examples of square root calculation:

```
>>> 16**0.5 4.0 >>> import math >>> math.sqrt(16) 4.0
```

The math module allows you to do a number of useful operations:

```
>>> math.log(16, 2) 4.0 >>> math.cos( 0 ) 1.0
```

Note that you only need to execute the import command once after you start IDLE; however you will need to execute it agin if you *restart* the shell, as restarting resets everything back to how it was when you opened IDLE. Don't worry too much about this right now; we'll cover it more in depth soon!

Exercise

(this is just for practice, solutions will not be graded or collected in class) Use IDLE to calculate:

- 6+4*10
- (6+4)*10 (Compare this to #1, and note that Python uses parentheses just like you would in normal math to determine order of operations!)
- 23.0 to the 5th power
- Positive root of the following equation: 34*x^2 + 68*x - 510 Recall: a*x^2 + b*x + c x1 = (- b + sqrt (b*b - 4*a*c)) / (2*a)

Sources

[1] Sarina Canelake. *6.189 A Gentle Introduction to Programming Using Python.* January IAP 2011. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

[2] Ana Bell, Eric Grimson, and John Guttag. *6.0001 Introduction to Computer Science and Programming in Python.* Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

Course 2 : Data structure

1. Types conteneurs :

Un conteneur est un objet composite destiné à contenir d'autres objets : nous distinguons les séquences (comprenant les types chaîne, liste, tuple), les tableaux associatifs, les ensembles (comprenant les types set et frozenset), les dictionnaires et les fichiers textuels. Les conteneurs sont des objets itérables.

Deux classements sont possibles :

- Mutables (modification autorisée) et non mutables (modification non autorisée)
- Ordonnés et non ordonnés

1.2.1. Les séquences :

Une séquence est un conteneur ordonné d'éléments indexés par des entiers indiquant leurs positions dans le conteneur. Les indices commencent par 0.si S une séquence :

- S[i] retourne i+1eme élément de S
- S [: a] sous séquence d'éléments de début jusqu'à l'élément d'indice a exclu
- S [a :] sous séquence d'éléments de l'indice a inclus jusqu'à la fin du séquence
- S [:] toute la séquence
- S [a : b] sous séquence d'éléments allant de l'indice a inclus jusqu'à l'élément d'indice b exclu
- S [a : b :c] sous séquence d'éléments de l'indice a inclus jusqu'à l'élément d'indice b exclu par pas égale à c.

a. Les listes :

Une liste est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes séparés par des virgules et définis entre crochet, les indices de liste commencent par 0, chaque liste peut être découpée, concaténée,…

```
>>> l=['info','math','phys','STI','ang','fran']
>>> l
['info', 'math', 'phys', 'STI', 'ang', 'fran']
>>> type(l)
<class 'list'>
>>> l[0]
'info'
>>> l[-1]
```

```
'fran'
>>> len(l)
6
>>> l[2:4]
['phys', 'STI']
>>> 2*l[0:]
['info', 'math', 'phys', 'STI', 'ang', 'fran', 'info', 'math', 'phys', 'STI', 'ang', 'fran']
>>> ['MP','PC'] +l
['MP', 'PC', 'info', 'math', 'phys', 'STI', 'ang', 'fran']
>>> l.append(6) #ajout à la fin du liste
>>> l
['info', 'math', 'phys', 'STI', 'ang', 'fran', 6]
>>> l.insert(0,'FSM') #ajout à la position 0 dans la liste
>>> l
['FSM', 'info', 'math', 'phys', 'STI', 'ang', 'fran', 6]
>>> l.remove(6) #supprime la première valeur 6 trouvée de la liste
>>> l
['FSM', 'info', 'math', 'phys', 'STI', 'ang', 'fran']
>>> l.pop() #supprime et retourne le dernier élément de la liste
'fran'
>>> l
['FSM', 'info', 'math', 'phys', 'STI', 'ang']
>>> l.pop(3)#supprime et retourne l'élément d'indice 3 de la liste
'phys'
>>> l
['FSM', 'info', 'math', 'STI', 'ang']
>>> del(l[4])#supprime l'élément d'indice 4 de la liste
>>> l
['FSM', 'info', 'math', 'STI']
>>> 'info' in l #tester l'existance de 'info' dans la liste
True
>>> l1=list('bonjour') #transformation d'une chaîne de caractère à une liste
>>> l1
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

La fonction range() génère un itérateur sur une progression arithmétique.

```
>>> l2=list(range(0,10))
>>> l2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l3=list(range(3,100,30))
>>> l3
[3, 33, 63, 93]
>>> l4=list(range(10,1,-1))
>>> l4
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>> l4.reverse () #permet d'inverser la liste
>>> l4
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l4.count (6) #calculer nombre d'occurrence de l'élément 6 dans l4
1
>>> l4.index(6) #indique la position de 6 dans l4
4
>>> l4.extend([22,77,88]) #ajout des éléments 22,77,88 à la fin de l4
>>> l4
[2, 3, 4, 5, 6, 7, 8, 9, 10, 22, 77, 88]
```

Listes définies par compréhension

On peut aussi définir une liste par compréhension avec la syntaxe :

[fontion(x) for x in liste if test(x)]

- Exemples :

```
>>> [2*x for x in range(10)]

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> [x for x in range(10) if x>5]
```

```
 [6, 7, 8, 9]

>>> [2*x for x in range(10) if x>5]

[12, 14, 16, 18]

>>> [2*x for x in range(10) if 2*x>5]

[6, 8, 10, 12, 14, 16, 18]

>>> [x for x in range(10) if x>5 and x<8]

[6, 7]

>>> [x+1 for x in [2*x for x in range(10)]]

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

>>> [2*x for x in [x+1 for x in range(10)]]

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

>>> [i*2 for i in range(10)]

[0, 2, 4, 6, 8, 10 ,12 ,14 ,16 ,18]

>>> [i for i in range(10) if i % 2 == 0]

[0, 2, 4, 6, 8]
```

- Plus compliqué, obtenir tous les nombres premiers inférieur ou égaux à 50 :

```
>>>noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>>primes = [x for x in range(2, 50) if x not in noprimes]
>>>primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

a. Les chaînes de caractères :

Une chaîne de caractère est une séquence non modifiable composée de caractères unicodes entre simple ou double côtes, elle admet le même principe d'indexation que les listes.

```
>>> ch='bonjour'
>>> type(ch)
<class 'str'>
>>> ch1=''
>>> ch2=""
>>> ch1==ch2
True
>>> ch3="  "
>>> ch3==ch2
False
>>> ch[0]
'b'
>>> ch[-1]
'r'
>>> len(ch)
7
>>> ch[:3]
'bon'
>>> ch[3:]
'jour'
>>> ch[2:5]
'njo'
>>> ch[::-1]
'ruojnob'
>>> ch[0]='A'
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    ch[0]='A'
TypeError: 'str' object does not support item assignment
>>> ch.isupper() #tester si la chaîne contient que des lettres majuscule
False
```

```
>>> ch.islower()#tester si la chaîne contient que des lettres miniscules
True
>>> ch.istitle()#retourne true si seul la premier lettre de chaque mot de la chaîne commence par majuscule
False
>>> ch.istitle()#retourne true si seul la premier lettre de chaque mot de la chaîne est en majuscule
False
>>> ch.isalpha() #retourne True si la chaîne ne contient que des caractère alphabétique
True
>>> ch.isalnum() #retourne True si la chaîne ne contient que des caractère alphanumérique
True
>>> ch.isnumeric() #retourne True si la chaîne ne contient que des caractère numérique
False
>>> ch.isdigit() #retourne True si la chaîne ne contient que des caractère numérique
False
>>> ch.isspace() #retourne True si la chaîne ne contient que des espaces
False
a='bonjour tout les mondes'
>>> a.split() #convertir a en une liste de mots
[' bonjour', 'tout', 'les', 'mondes']
```

a. Les tuples :

Un tuple (ou n-uplet) est une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes, les éléments sont séparés par des virgules, et entourés de parenthèses. L'indexage des tuples s'utilise comme celui des listes.

```
# définition d'un tuple
>>> t = ("a", "b", "mpilgrim", "z", "example")
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]
```

```
'a'
>>> t[-1]
'example'
>>> t[1:3]
('b', 'mpilgrim')
>>>u=t,(1,2,3,4,5)  #les tuples peuvent être imbriqués
>>> u
(('a', 'b', 'mpilgrim', 'z', 'example'), (1, 2, 3, 4, 5))
```

Un problème particulier (tuple 0 ou 1 élément) :

- Les tuples vides construits avec des parenthèses vides.
- Un tuple avec un élément est construit en faisant suivre une valeur d'une virgule.

```
Tvide=() #tuple vide
>>> Tvide
()
>>> T='bonjour', #tuple avec 1 élément notez bien la virgule en fin de ligne
>>> len(Tvide)
0
>>> len(T)
1
>>> T
('bonjour',)
```

Les tuples n'ont pas de méthodes pour ajouter, modifier ou supprimer ses éléments.

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    1
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      2
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
```

```
>>> t.index("example")
4
>>> "z" in t          4
True
```

Remarque :

- Les tuples n'ont pas de méthodes append ou extend pour ajouter d'élément.

- Les tuples n'ont pas de méthodes remove ou pop pour enlever d'éléments.

- Les tuples permet d'utiliser **in** pour vérifier l'existence d'un élément.

Mais à quoi servent donc les tuples ?

- Les tuples sont plus rapides que les listes. Si vous définissez un ensemble constant de valeurs et que tout ce que vous allez faire est le parcourir, utilisez un tuple au lieu d'une liste.

- Votre code est plus sûr si vous «protégez en écriture» les données qui n'ont pas besoin d'être modifiées. Utiliser un tuple à la place d'une liste revient à avoir une assertion implicite que les données sont constantes et que des mesures spécifiques sont nécessaires pour modifier cette définition.

2.2.2. Les ensembles :

a. Définition

Un ensemble est une collection d'éléments distincts, non ordonnés et non dupliqués. On peut créer un ensemble en listant tous ses éléments, séparés par des virgules, et en délimitant le tout avec des accolades.

```
# Définition
numbers = {42, -2, 0, 7, 11}
# seconde façon de définir un set avec doubles ( )
set1 = set(['b','c','d','c']) # {'c', 'b', 'd'}
print(numbers)          # {0, 42, 11, -2, 7}
print(len(numbers))     # 4
print(60 in numbers)    # False
print(type(numbers))    # < class ' set '>
for element in numbers :
        print ( element )
```

Remarque :

- La fonction set() peut être utilisée pour créer des ensembles. Notez que pour créer un ensemble vide, {} ne fonctionne pas, cela créé un dictionnaire vide. Utilisez plutôt set().

- On ne peut par contre pas accéder aux éléments d'un ensemble à partir d'un indice, puisqu'il s'agit d'une collection non ordonnée d'éléments, et on ne peut donc parcourir les éléments d'un ensemble qu'avec la boucle for.

b. Opérations sur les ensembles

- Modification d'un ensemble par l'ajout/suppression d'éléments avec les fonctions add et remove.

- Application de la fonction sur la variable contenant la liste :

Nom de la variable, suivi d'un point (.) puis du nom de la fonction à appeler, avec ses éventuels paramètres.

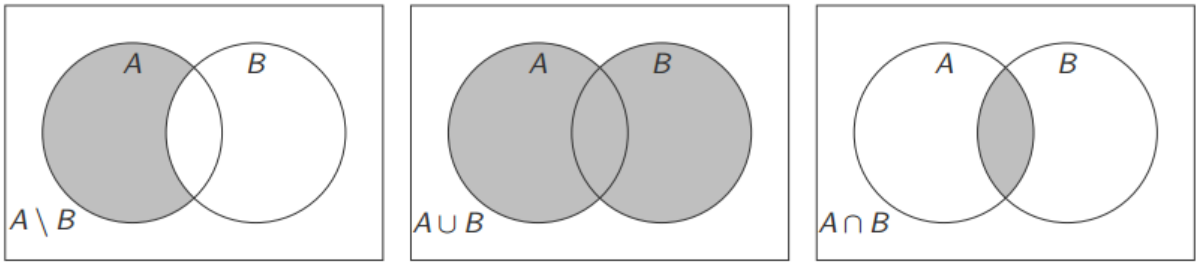
```
S = {1, 2, 3, 4}
S.add(5)                    # ajouter 1 seulélément→{1, 2, 3, 4, 5}
S.remove(5)#  détruire 1 seulélément→{1, 2, 3, 4}
S.discard(4)# remove, mais pas erreur si l'élément est absent→{1, 2, 3}
S.clear()# détruire tous les éléments→{ }
```

c. Opération ensembliste

- Trois opérations ensemblistes de base à savoir :

- Différence
- Union
- Intersection d'ensembles

- Ces fonctions créent toutes des nouveaux ensembles :



d.Le type frozenset

Un frozenset est simplement un set immutable, c'est-à-dire non modifiable. On crée un objet de type frozenset avec la fonction frozenset(iterable).

```
fset = frozenset( {'x','y','z','x'} ) # ou...
fset = frozenset( ('x','y','z','x') ) # identique
type(fset)   # => type builtins.frozenset
```

Remarque:

Tout ce que l'on a vu concernant les sets est applicable, à l'exception de ce qui a trait à la modification. Les frozensets occupent moins d'espace en mémoire que les sets, et leur traitement est plus efficace.

Les ensembles en compréhension :

```
>>> {n for n in range(5)}
{0, 1, 2, 3, 4}
```

1.2.1. Les dictionnaires :

Un dictionnaire (tableau associatif) est un type de données permettant de stocker des couples **clé : valeur**, avec un accès très rapide à la valeur à partir de la clé, la clé ne pouvant être présent qu'une seule fois dans le tableau**.**

Caractéristiques :

- L'opérateur d'appartenance d'une clé (in)
- La fonction taile (len()) donnant le nombre des couples stockés
- Permet de retrouver un objet par sa clé
- Il est itérable (on peut le parcourir)
- Non ordonné
- Mutable

```
#définition d'un dictionnaire
>>> dic={'1':'un','2':'deux','3':'trois'}
>>> dic
{'1': 'un', '3': 'trois', '2': 'deux'}
```

```
>>> #ou élément aprés élément
>>> dic={}
>>> dic['1']='un';dic['2']='deux';dic['3']='trois'
>>> dic
{'1': 'un', '3': 'trois', '2': 'deux'}
>>> len(dic) #taille de dictionnaire
3
>>>type(dic)
<class 'dict'>
>>> '1' in dic
True
>>> '4' in dic
False
>>> dic={'1':'un','2':'deux','3':'trois'}
>>> dic
{'1': 'un', '3': 'trois', '2': 'deux'}
>>> dic[4]='quatre' #ajout élément
>>> dic
{'1': 'un', '3': 'trois', '2': 'deux', 4: 'quatre'}
>>> del(dic[4]) #suppression de l'élément 4:'quatre'
>>> dic
{'1': 'un', '3': 'trois', '2': 'deux'}
>>> val=dic.pop('3') #récupère val et supprime l'élément
>>> dic
{'1': 'un', '2': 'deux'}
>>> val
'trois'
>>> dic.update({'a':4,'b':6}) #ajout plusieurs élément
>>> dic
{'1': 'un', 'a': 4, '2': 'deux', 'b': 6}
>>> dic={'1':'un','2':'deux','3':'trois'}
>>> dic.keys() #donne les clefs d'un dictionnaire
dict_keys(['1', '3', '2'])
>>> dic.values()#donne les valeurs d'un dictionnaire
```

```
dict_values(['un', 'trois', 'deux'])
>>> dic.items() #fournit clefs et valeurs du dictionnaire sous forme de couples
dict_items([('1', 'un'), ('3', 'trois'), ('2', 'deux')])
>>> list(dic.keys()) #convertir les clefs en une list
['1', '3', '2']
>>> tuple(dic.values()) #convertir les valeurs en tuple
('un', 'trois', 'deux')
>>> dic.clear() #supprimer tous les éléments
>>> dic
{}
```

Les dictionnaires en compréhension :

```
>>> {x : x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Course 3: Recursion

I. Introduction :

La récursivité est une méthode de programmation puissante qui permet d'exprimer d'une manière élégante la solution de problèmes. Une fonction récursive est une fonction dont la définition contient un (ou plusieurs) appel à elle-même.

Exemple 1 : Le calcul de la factorielle

Pour n≥0, n !=$\prod_{n=1}^{n} i$

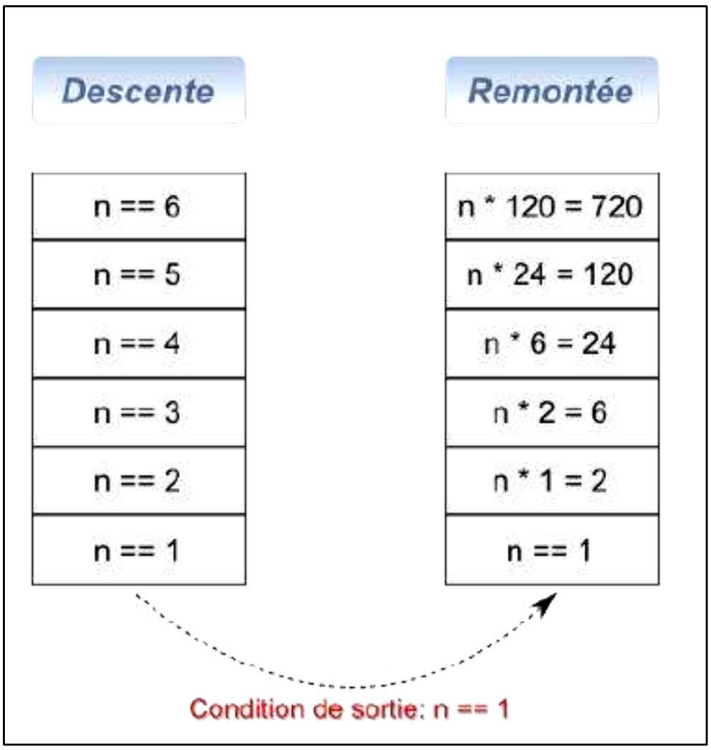
Solution itérative	Solution récursive

def fact(n) :
f=1
for i in range (1,n+1):
f=f*i
return(f)
 	
print(fact(10))
3628800
	def fact(n) :
	if n==0:
	return(1)
	else:
	return(n*fact(n-1))
	
	print(fact(10))
	3628800

Exécution :

Les appels successifs d'une fonction récursive sont stockés dans une pile : c'est la phase de descente. Une fois que la close d'arrêt est demandée, les appels sont ensuite désempilés jusqu'à arriver à l'appel initial : c'est la phase de remontée.

Exemple pour le calcul de 6! :



Exemple 2:

Les suites récurrentes sont des exemples classiques.

On considère la suite Un définie par :

$$\begin{cases} U_0 = 1 \\ Un+1 = \frac{1}{2}(Un + \frac{2}{Un}) \end{cases}$$

Solution itérative	Solution récursive

def U(n) :
x=1
for i in range (1,n+1):
x=1/2*(x+2/x)
return(x)
 	
print(U(10))
1.41421356
	def U(n) :
	if n==0:
	return(1)
	else:
	return(1/2*(U(n-1)+2/U(n-1)))
	
	print(U(10))
	1.41421356

- Ces exemples montrent les trois grands principes d'une fonction récursive :
  - Une fonction récursive doit comporter une close d'arrêt qui consiste en la donnée d'un ou plusieurs cas de base : c'est le type de la sortie de cette close qui va déterminer le type de la sortie de la fonction.

- Une fonction récursive doit appeler la close d'arrêt au bout d'un nombre fini d'étapes (ce qui assure la terminaison de l'algorithme).
- Une fonction récursive s'appelle elle-même, mais avec différentes valeurs de (ou des) l'argument(s).

❖ La structure générale est donc la suivante :

```
1 def fonction_recursive(paramètre):
2     if ... :      # close d'arrêt
3         return ...  # cette ligne détermine le type de la
              variable de sortie
4     else:
5         return ... # instructions contenant un appel à
              fonction_recursive
```

➢ On remarque que l'instruction ***return*** doit être présente au moins deux fois : une fois pour la close d'arrêt et une fois pour l'appel récursif.

I. Exercices :

Exercice 1 :

La suite de Fibonacci est définie par $f_0 = f_1 = 1$ et pour tout naturel $n \geq 2$ par :

$$f_n = f_{n-1} + f_{n-2}$$

Écrire une fonction python récursive f (n) calculant le terme d'indice n de cette suite.

Exercice 2 :

Ecrire une fonction récursive en Python permettant de calculer la fonction f(n)=3*n c-à-d les multiples de 3.

Exercice 3 :

Ecrire une fonction récursive en Python qui renvoie la somme de n premiers entiers.

Exercice 4 : Calcul de PGCD

Ecrire une fonction Python qui permet de calculer le pgcd de deux entiers selon l'algorithme d'Euclide.

$$pgcd\ (a,b) = b\ \ si\ a\ est\ un\ multiple\ de\ b$$

$$Sinon\ \ pgcd\ (a,b) = pgcd\ (b, a\ mod\ b).$$

Exercice 5 :

On considère la suite Un suivante, qui calcule une approximation de $\sqrt{3}$

$$\begin{cases} U_0 = 2 \\ Un = \frac{1}{2}(U(n-1) + \frac{3}{Un-1}) \end{cases}$$

Ecrire une fonction récursive en Python qui renvoie la valeur u(n).

Exercice6 :

Un palindrome est un mot dont les lettres lues de gauche à droite sont les mêmes que celles lues de droite à gauche. Les mots radar, elle, été, ici sont des palindromes.

Ecrire une fonction récursive qui teste si un mot est un palindrome.

Exercice 7: Algorithmes récursifs sur les listes

1. Donnez une fonction récursive de l'inversion de l'ordre des éléments d'une liste.
2. Ecrire une fonction récursive qui teste si deux listes d'entiers sont identiques.
3. Ecrire une fonction récursive qui compte le nombre d'occurrences d'un élément dans une liste.