PYTHON NOTES

L'indentation est très importante, elle permet de délimiter des blocs de codes (comme {} dans les autres langages C, C++, JAVA) et est essentiel pour l'interpréteur Python

```
import sys # Import du Module (librairie) sys

# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
    #Générer une documentation:
    """
    Returns the string 's' repeated 3 times.
    If exclaim is true, add exclamation marks.
    """
    result = s + s + s # can also use "s * 3" which is faster (Why?)
    result = s*3
    if exclaim:
        result = result + '!!!'
    return result

# Syntaxe de déclaration d'une fonction
def main(name):
# Syntaxe d'une boucle If
    if name == 'Wissdom':
        print repeat(name,False) + '?'
    else:
        print repeat(name,True)
if __name__ == '__main__':
    # Code à exécuter lors de l'exécution du programme
    main(sys.argv[1]) # sys.argv contient les argument d'exécution du programme comme en C
ou en JAVA
    sys.exit(0)
```

Pour avoir de l'aide sur des fonctions, des variables d'un module :

Dans l'interpréteur de commande Python :

>>import module as mod

>>help(mod) ou ***help(mod.func)***

Pour une version plus courte :

>>dir(mod) ou ***dir(mod.func)***

Boucle for :

```
for i in range(1,100,1):
#Boucle de 1 à 100 avec un pas de 1 range(debut,fin,pas)
#Suite d'instructions#
```

Boucle If:

```
if name == 'Wissdom':
print(repeat(name,False) + '?')
elif name == 'Lux':
print(repeat(name,False)+ 'UUU')
else:
print(repeat(name,True))
```

Pour écrire du code sur plusieurs lignes on peut utiliser des **{}** ou des **()** ou encore des **[]**

```
# add parens to make the long-line work:
text = ("%d little pigs come out or I'll %s and %s and %s" %
(3, 'huff', 'puff', 'blow down'))
```

Exception TRY/CATCH

```
try:
    5/0
except:
    print("Please don't do that")
```

Les STRING (Module *STR*)

```
# Pour declarer une chaîne de caractère (String)

 s = 'hi'

 print s[1]          ## i

 print len(s)        ## 2

 print s + ' there'  ## hi there

 c='J\'ai %d ans et je m\'appelle %s' %(24,'Wissam')

 print(c)   ## J'ai 24 ans et je m'appelle Wissam
```

Contrairement à d'autres langages comme JAVA ou C, il n'y a pas d'héritage entre les types de variables (c-à-d ***INT*** n'hérite pas de ***DOUBLE*** qui n'hérite pas de ***STRING)***, il n'y a pas de conversion implicite : le '+' ne convertit pas automatiquement les nombres ou d'autres types de variables en ***STRING***. La fonction ***str()*** convertit les valeurs en une forme ***STRING***, pour pouvoir ensuite les concaténer avec d'autre chaîne de caractère. De plus les chaînes de caractères sont immuables on ne peut pas faire s[0]='l' une fois que s est déclarée.

```
pi = 3.14
##text = 'The value of pi is ' + pi      ## NO, does not work
text = 'The value of pi is '  + str(pi)  ## yes
```

Pour que des caractères spéciaux ne soient pas interprétés dans une chaîne caractère, on utilise les « ***Raw*** » Strings.

```
# Creation of a raw String

 raw = r'this\t\n and that'

 print raw     ## this\t\n and that

 multi = """It was the best of times.It was the worst of times."""

 print multi ## It was the best of times.It was the worst of times.
```

On peut aussi créer des string « ***Unicode »*** :

```
ustring = u'A unicode \u018e string \xf1'

## (ustring from above contains a unicode string)

s = ustring.encode('utf-8')

print('utf8 '+s)

'A unicode \xc6\x8e string \xc3\xb1'  ## bytes of utf-8 encoding

t = unicode(s, 'utf-8')             ## Convert bytes back to a unicode string

t == ustring                       ## It's the same as the original, yay!

print(ustring)
```

Quelques fonctions à utiliser avec des *STR(String)* :

- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

STR SLICES*:*

```
s='Hello'
print(s[1:3]) # Affiche el      chars starting at index 1 and extending up to but not
including index 3
print(s[1:]) # Affiche ello     = print(s[1:len(s)-1])  omitting either index defaults to
the start or end of the string
print(s[:]) # Affiche Hello     = print(s)  omitting both always gives us a copy of the
whole thing (this is the pythonic way to copy a sequence like a string or list)
print(s[1:100]) # Affiche ello  si len(s)<=100 equivalent to print(s[1:len(s)-1])
print(s[-1]) # Affiche o    = print(s[len(s)-1])    last char (1st from the end)
print(s[-4]) # Affiche e    = print(s[len(s)-4]) 4th from the end
print(s[:-3]) # Affiche He  = print(s[0:len(s)-1-3]) going up to but not including the
last 3 chars.
print(s[-3:]) # Affiche llo = print(s[3,len(s)-1]) starting with the 3rd char from the end
and extending to the end of the string
```

Les LISTES

```
colors = ['red', 'blue', 'green']
print colors[0]    ## red
print colors[2]    ## green
print len(colors)  ## 3

b = colors   ## Does not copy the list
print('list b'+str(b)) ## [‘red’, ‘blue’, ‘green’]

[a,b,c]=[1,2,5] ## a=1, b=2 et c=5 mais l=([a,b,c]=[1,2,5]) ne marche pas
print c  ## 5

squares = [1, 4, 9, 16]
sum = 0
for num in squares:
sum += num ## 1+4+9+16
print sum  ## 30

l = ['larry', 'curly', 'moe']
for c in l:
print(c) ## Affiche larry curly moe

if 'curly' in l:
print 'curl est dans l' ## Affiche curl est dans l
range(10) ## créer une liste [1,2,3,4,5,6,7,8,9]

for i in range(10): ## boucle de 1 à 10
        print(i)

for i in xrange(10): ## boucle de 1 à 10, plus rapide que range(10)
        print(i)
```

List Methods : Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() -- sorts the list in place (does not return it). (The sorted() function shown below is preferred.)
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Exemple:

```
list = ['larry', 'curly', 'moe']
list.append('shemp')         ## append elem at end
list.insert(0, 'xxx')        ## insert elem at index 0
list.extend(['yyy', 'zzz'])  ## add list of elems at end
print list  ## ['xxx', 'larry', 'curly', 'moe', 'shemp', 'yyy', 'zzz']
print list.index('curly')    ## 2

list.remove('curly')         ## search and remove that element
list.pop(1)                  ## removes and returns 'larry'
print list  ## ['xxx', 'moe', 'shemp', 'yyy', 'zzz']
list = []          ## Start as the empty list
list.append('a')   ## Use append() to add elements
list.append('b')
```

LIST SLICES

```
list = ['a', 'b', 'c', 'd']
print list[1:-1]   ## ['b', 'c']
list[0:2] = 'z'    ## replace ['a', 'b'] with ['z']
print list         ## ['z', 'c', 'd']
```

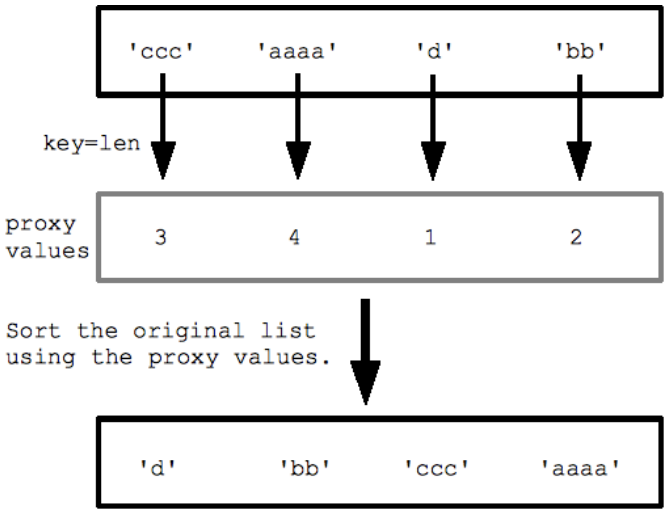
SORT LIST

```
a = [5, 1, 4, 3]
print sorted(a)  ## [1, 3, 4, 5]
print a  ## [5, 1, 4, 3]

strs = ['aa', 'BB', 'zz', 'CC']
print sorted(strs)  ## ['BB', 'CC', 'aa', 'zz'] (case sensitive)
print sorted(strs, reverse=True)   ## ['zz', 'aa', 'CC', 'BB']
```

tri personnalisé avec clé

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)  ## ['d', 'bb', 'ccc', 'aaaa']
```

Pour des tris encore plus personnalisés, on peut utiliser l'option 'key=' qui permet de spécifier une 'key' fonction qui transforme chaque élément de la liste avant de comparer.



```
strs = ['aa','zz','BB','CC']## "key" argument specifying str.lower function to use for
sorting
print sorted(strs, key=str.lower)  ## ['aa', 'BB', 'CC', 'zz']
```

On peut aussi utiliser ses propres fonctions à condition qu'elle retourne des valeurs

```
## Say we have a list of strings we want to sort by the last letter of the string.
strs = ['xc', 'zb', 'yd' ,'wa']

## Write a little function that takes a string, and returns its last letter.
## This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
    return s[-1]
## Now pass key=MyFn to sorted() to sort by the last letter:
    print sorted(strs, key=MyFn)  ## ['wa', 'zb', 'xc', 'yd']
```

To use key= custom sorting, remember that you provide a function that takes one value and returns the proxy value to guide the sorting. There is also an optional argument "cmp=cmpFn" to ***sorted()*** that specifies a traditional two-argument comparison function that takes two values from the list and returns negative/0/positive to indicate their ordering. The built in comparison function for strings, ints, ... is cmp(a, b), so often you want to call cmp() in your custom comparator. The newer one argument key= sorting is generally preferable.

Alternative au ***sorted()*** :

```
alist.sort()            ## correct
alist = blist.sort()    ## NO incorrect, sort() returns None
```

Les TUPLES

Un tuple est un groupe de taille fixe composé d'éléments, comme des coordonnées (x,y). Ils sont comme les listes sauf qu'ils sont immuables(inaltérables) et ne peuvent changer de nombre d'éléments. Ils ont un rôle de structure en Python et peuvent être considérés comme « a convenient way to pass around a little logical, fixed size bundle of values ».

```
tuple = () ##empty tuple
tuple = ('hi',) ## size-1 tuple, la virgule est importante car permet de distinguer un
tuple d'un simple bloc d'instructions délimité par des ()
tuple = (1, 2, 'hi')
print len(tuple)  ## 3
print tuple[2]    ## hi
tuple[2] = 'bye'  ## NO, tuples cannot be changed
tuple = (1, 2, 'bye')  ## this works

(x, y, z) = (42, 13, "hike")
print z  ## hike
(err_string, err_code) = Foo()  ## Error Foo is not defined
```

List Comprehensions

Pour faire des opérations sur les listes en écrivant le moins de code possible.

```
nums = [1, 2, 3, 4]
squares = [ n * n for n in nums ]   ## [1, 4, 9, 16]
```

La syntaxe est *[expr for var in list]* : la boucle *for var in list* ressemble à une boucle for normale, mais il n'y a pas les « : ». *expr* est évaluée une fois pour chaque élément afin d'obtenir sa nouvelle valeur dans la nouvelle liste.

Autre exemple avec des strings :

```
strs = ['hello', 'and', 'goodbye']
shouting = [ s.upper() + '!!!' for s in strs ] ## ['HELLO!!!', 'AND!!!', 'GOODBYE!!!']
```

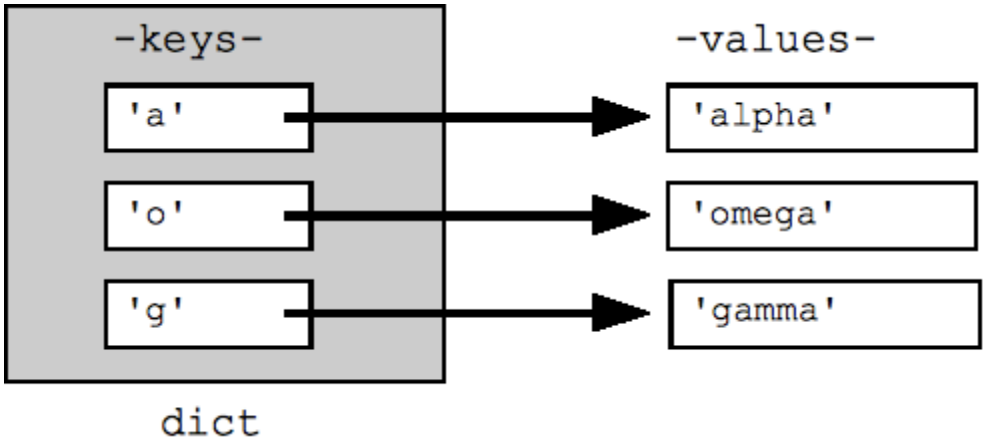
On peut aussi ajouter de test de conditions avec le ***if***. Le if test est évalué pour chaque élément, et n'inclut que les éléments où la condition est vraie :

```
## Select values <= 2
nums = [2, 8, 1, 6]
small = [ n for n in nums if n <= 2 ]  ## [2, 1]

## Select fruits containing 'a', change to upper case
fruits = ['apple', 'cherry', 'bannana', 'lemon']
afruits = [ s.upper() for s in fruits if 'a' in s ] ## ['APPLE', 'BANNANA']
```

Les DICTIONNAIRES ou DICT HASH TABLE

Un dictionnaire ou dict est en python est une structure de tableau de hashage avec clé-valeur.
Pour déclarer un dictionnaire : ***dict = {key1 :value1, key2 :value2,…}***
key doit être immuable, elle peut être un ***string, tuple*** ou ***integer***.

```
## Can build up a dict by starting with the the empty dict {}
## and storing key/value pairs into the dict like this:
## dict[key] = value-for-that-key
dict = {} ## empty dict
dict['a'] = 'alpha'
dict['g'] = 'gamma'
dict['o'] = 'omega'
dict[1] = 'zeta'
dict[0] = 17

print dict  ## {0: 17, 'a': 'alpha', 1: 'zeta', 'o': 'omega', 'g': 'gamma'}
print dict['a'] ## Simple lookup, returns 'alpha'
print dict[4]     ## KeyError : 4  la clé 4 n'est pas défini
dict['a'] = 6       ## Put new key/value into dict
'a' in dict         ## True
## print dict['z']                  ## Throws KeyError
if 'z' in dict: print dict['z']     ## Avoid KeyError
print dict.get('z')  ## None (instead of KeyError)
```



Une boucle sur un dictionnaire par défaut boucle sur les clés du dictionnaire. Les clés apparaissent dans un ordre arbitraire. On peut boucler sur la liste des clés ou la liste des valeurs ou les deux avec les fonctions :
`dict.keys()` retourne la liste des clés
`dict.values()` retourne la liste des valeurs
`dict.items()` retourne une liste de doublets (clés,valeurs), méthode la plus efficace pour examiner les couples clés-valeurs dans un dictionnaire

```
dict={'a':'alpha', 'o':'omega', 'g':'gamma'}
## By default, iterating over a dict iterates over its keys.
## Note that the keys are in a random order.
for key in dict: print key ## prints a g o

## Exactly the same as above
for key in dict.keys(): print key

## Get the .keys() list:
print dict.keys()  ## ['a', 'o', 'g']

## Likewise, there's a .values() list of values
print dict.values()  ## ['alpha', 'omega', 'gamma']

## Common case -- loop over the keys in sorted order,
## accessing each key/value
for key in sorted(dict.keys()):
print key, dict[key]

## .items() is the dict expressed as (key, value) tuples
print dict.items()  ##  [('a', 'alpha'), ('o', 'omega'), ('g', 'gamma')]

## This loop syntax accesses the whole dict by looping
## over the .items() tuple list, accessing one (key, value)
## pair on each iteration.
for k, v in dict.items(): print k, '>', v ## a > alpha    o > omega     g > gamma
```

Tips : En terme de performance le dictionnaire est l'un des meilleurs outils. Il est préférable de l'utiliser dès qu'on le peut, il permet d'organiser simplement des données dans une structure cohérente.

Pour formater un dictionnaire dans un string, on utilise l'opérateur %.

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count)d copies of %(word)s' % hash  # %d for int, %s for string
# 'I want 42 copies of garfield'
print(s)
```

Effacer des variables :

L'opérateur ***del*** permet d'effacer n'importe quelle variable, la variable n'est alors plus définie. L'opérateur ***del*** peut aussi être utiliser dans des slices pour effacer une partie de liste ou des entrée d'un dictionnaire.

```
var = 6
del var  # var no more!

list = ['a', 'b', 'c', 'd']
del list[0]     ## Delete first element
del list[-2:]   ## Delete last two elements
print list      ## ['b']

dict = {'a':1, 'b':2, 'c':3}
del dict['b']   ## Delete 'b' entry
print dict      ## {'a':1, 'c':3}
```

FILES

The ***open()*** function opens and returns a file handle that can be used to read or write a file in the usual way. The code ***f = open('name', 'r')*** opens the file into the variable f, ready for reading operations, and use ***f.close()*** when finished. Instead of ***'r'***, use ***'w'*** for writing, and 'a' for append. The special mode ***'rU'*** is the "Universal" option for text files where it's smart about converting different line-endings so they always come through as a simple ***'\n'***. The standard for-loop works for text files, iterating through the lines of the file (this works only for text files, not binary files). The for-loop technique is a simple and efficient way to look at all the lines in a text file:

```
# Echo the contents of a file
f = open('foo.txt', 'rU')
for line in f:  ## iterates over the lines of the file
    print line, ## trailing , so print does not add an end-of-line char
                ## since 'line' already includes the end-of line.
f.close()
```

Reading one line at a time has the nice quality that not all the file needs to fit in memory at one time -- handy if you want to look at every line in a 10 gigabyte file without using 10 gigabytes of memory. The ***f.readlines()*** method reads the whole file into memory and returns its contents as a list of its lines. The ***f.read()*** method reads the whole file into a single string, which can be a handy way to deal with the text all at once, such as with regular expressions we'll see later.

For writing, ***f.write(string)*** method is the easiest way to write data to an open output file. Or you can use ***print*** with an open file, but the syntax is nasty: ***print >> f, string***. In python 3, the print syntax will be fixed to be a regular function call with a file= optional argument:
print(string,file=f).

FILES UNICODE

The "codecs" module provides support for reading a unicode file.

```
import codecs

f = codecs.open('foo.txt', 'rU', 'utf-8')
for line in f:# here line is a *unicode* string,
```

For writing, use ***f.write()*** since print does not fully support unicode.

Les EXPRESSIONS RÉGULIÈRES

Le module **re** permet de gérer les expressions régulières dans python. Une expression régulière se déclare ainsi :

```
match = re.search(pat, str)
```

La méthode ***re.search()*** prend comme motif une expression régulière et un string et cherche ce motif dans le string. Si la rechercher est un succès, elle renvoie un objet de type **`match`** sinon un objet de type ***null.***
Comment exploiter le match :

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
if match:
print 'found', match.group() ## 'found word:cat'
else:
    print 'did not find'
```

Basic Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

- a, X, 9, < -- ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ^ $ * + ? { [ ] \ | ( ) (details below)
- . (a period) -- matches any single character except newline '\n'
- \w -- (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.
- \b -- boundary between word and non-word
- \s -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f]. \S (upper case S) matches any non-whitespace character.
- \t, \n, \r -- tab, newline, return
- \d -- decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)
- ^ = start, $ = end -- match the start or end of the string
- \ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Règles sur les expressions régulières :

- La recherche se fait du début jusqu'à la fin du string, elle s'arrête au premier match trouvé
- Tous les motifs doivent être match, mais pas tout le str sera exploré
- Si `match = re.search(pat, str)` est un succès, match n'est pas None et match.group() est le texte qui a matché

```
## Search for pattern 'iii' in string 'piiig'.
## All of the pattern must match, but it may appear anywhere.
## On success, match.group() is matched text.
match = re.search(r'iii', 'piiig') =>  found, match.group() == "iii"
match = re.search(r'igs', 'piiig') =>  not found, match == None

## . = any char but \n
match = re.search(r'..g', 'piiig') =>  found, match.group() == "iig"

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') =>  found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') =>  found, match.group() == "abc"
```

Les Répétitions dans le pattern:

- + -- 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
- -- 0 or more occurrences of the pattern to its left
- ? -- match 0 or 1 occurrences of the pattern to its left

```
## i+ = one or more i's, as many as possible.
match = re.search(r'pi+', 'piiig') =>  found, match.group() == "piii"

## Finds the first/leftmost solution, and within it drives the +
## as far as possible (aka 'leftmost and largest').
## In this example, note that it does not get to the second set of i's.
match = re.search(r'i+', 'piigiiii') =>  found, match.group() == "ii"

## \s* = zero or more whitespace chars
## Here look for 3 digits, possibly separated by whitespace.
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') =>  found, match.group() == "1 2   3"
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') =>  found, match.group() == "12  3"
match = re.search(r'\d\s*\d\s*\d', 'xx123xx') =>  found, match.group() == "123"

## ^ = matches the start of string, so this fails:
match = re.search(r'^b\w+', 'foobar') =>  not found, match == None
## but without the ^ it succeeds:
match = re.search(r'b\w+', 'foobar') =>  found, match.group() == "bar"

str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'\w+@\w+', str)
if match:
print match.group()  ## 'b@google'
```

[] peuvent être utilizes pour indiqués un ensemble de caractères, par exemple [abc] match 'a' ou 'b' ou 'c'. Les codes ***\w***, ***\s*** etc marchent à l'intérieur des **[]** sauf le point (**.**) qui veut juste dire signifier littéralement un point.

```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
    print match.group()  ## 'alice-b@google.com'
```

On peut utiliser aussi le **–** pour indiquer une portée comme **[a-z]** qui permet de match toutes les lettres minuscules. Pour utiliser le **–** sans indiquer de portée, on met le **–** à la fin **[abc-]**. Le **^** en début de **[]** à l'effet inverse du **–** par exemple **[^ab]** signifie tout caractère sauf 'a' ou 'b'.

Group Extraction :

The "group" feature of a regular expression allows you to pick out parts of the matching text. Suppose for the emails problem that we want to extract the username and host separately. To do this, add parenthesis ( ) around the username and host in the pattern, like this: **r'([\w.-]+)@([\w.-]+)'**. In this case, the parenthesis do not change what the pattern will match, instead they establish logical "groups" inside of the match text. On a successful search, match.group(1) is the match text corresponding to the 1st left parenthesis, and match.group(2) is the text corresponding to the 2nd left parenthesis. The plain match.group() is still the whole match text as usual.

```
str = 'purple alice-b@google.com monkey dishwasher'
 match = re.search('([\w.-]+)@([\w.-]+)', str)
 if match:
    print match.group()   ## 'alice-b@google.com' (the whole match)
    print match.group(1)  ## 'alice-b' (the username, group 1)
    print match.group(2)  ## 'google.com' (the host, group 2)
```

A common workflow with regular expressions is that you write a pattern for the thing you are looking for, adding parenthesis groups to extract the parts you want.

findall

findall() is probably the single most powerful function in the re module. Above we used **re.search()** to find the first match for a pattern. **findall()** finds *all* the matches and returns them as a list of strings, with each string representing one match.

```
  ## Suppose we have a text with many email addresses
 str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'

 ## Here re.findall() returns a list of all the found email strings
 emails = re.findall(r'[\w\.-]+@[\w\.-]+', str) ## ['alice@google.com', 'bob@abc.com']
 for email in emails:
 # do something with each found email string
    print email
```

findall With Files

For files, you may be in the habit of writing a loop to iterate over the lines of the file, and you could then call **findall()** on each line. Instead, let **findall()** do the iteration for you -- much better! Just feed the whole file text into **findall()** and let it return a list of all the matches in a single step (recall that **f.read()** returns the whole text of a file in a single string):

```
# Open file
f = open('test.txt', 'r')
# Feed the file text into findall(); it returns a list of all the found strings
strings = re.findall(r'some pattern', f.read())
```

findall and Groups

The parenthesis ( ) group mechanism can be combined with **findall()**. If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, **findall()** returns a list of *tuples*. Each tuple represents one match of the pattern, and inside the tuple is the group(1), group(2) .. data. So if 2 parenthesis groups are added to the email pattern, then **findall()** returns a list of tuples, each length 2 containing the username and host, e.g. ('alice', 'google.com').

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
print tuples  ## [('alice', 'google.com'), ('bob', 'abc.com')]
for tuple in tuples:
   print tuple[0]  ## username
   print tuple[1]  ## host
```

Once you have the list of tuples, you can loop over it to do some computation for each tuple. If the pattern includes no parenthesis, then findall() returns a list of found strings as in earlier examples. If the pattern includes a single set of parenthesis, then findall() returns a list of strings corresponding to that single group. (Obscure optional feature: Sometimes you have paren ( ) groupings in the pattern, but which you do not want to extract. In that case, write the parens with a ?: at the start, e.g. (?:) and that left paren will not count as a group result.)

RE Workflow and Debug

Regular expression patterns pack a lot of meaning into just a few characters , but they are so dense, you can spend a lot of time debugging your patterns. Set up your runtime so you can run a pattern and print what it matches easily, for example by running it on a small test text and printing the result of findall(). If the pattern matches nothing, try weakening the pattern, removing parts of it so you get too many matches. When it's matching nothing, you can't make any progress since there's nothing concrete to look at. Once it's matching too much, then you can work on tightening it up incrementally to hit just what you want.

Options

The re functions take options to modify the behavior of the pattern match. The option flag is added as an extra argument to the **search()** or **findall()** etc., e.g. re.search(pat, str, re.IGNORECASE).

- IGNORECASE -- ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
- DOTALL -- allow dot (.) to match newline -- normally it matches anything but newline. This can trip you up -- you think .* matches everything, but by default it does not go past the end of a line. Note that \s (whitespace) includes newlines, so if you want to match a run of whitespace that may include a newline, you can just use \s*
- MULTILINE -- Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.

Greedy vs. Non-Greedy (optional)

This is optional section which shows a more advanced regular expression technique not needed for the exercises.

Suppose you have text with tags in it: foo and so on

Suppose you are trying to match each tag with the pattern '(<.*>)' -- what does it match first?

The result is a little surprising, but the greedy aspect of the .* causes it to match the whole 'foo and so on' as one big match. The problem is that the .* goes as far as is it can, instead of stopping at the first > (aka it is "greedy").

There is an extension to regular expression where you add a ? at the end, such as .*? or .+?, changing them to be non-greedy. Now they stop as soon as they can. So the pattern '(<.*?>)' will get just '' as the first match, and '' as the second match, and so on getting each <..> pair in turn. The style is typically that you use a .*?, and then immediately its right look for some concrete marker (> in this case) that forces the end of the .*? run.

The *? extension originated in Perl, and regular expressions that include Perl's extensions are known as Perl Compatible Regular Expressions -- pcre. Python includes pcre support. Many command line utils etc. have a flag where they accept pcre patterns.

An older but widely used technique to code this idea of "all of these chars except stopping at X" uses the square-bracket style. For the above you could write the pattern, but instead of .* to get all the chars, use [^>]* which skips over all characters which are not > (the leading ^ "inverts" the square bracket set, so it matches any char not in the brackets).

Substitution (optional)

The **re.sub(pat, replacement, str)** function searches for all the instances of pattern in the given string, and replaces them. The replacement string can include '\1', '\2' which refer to the text from group(1), group(2), and so on from the original matching text.

Here's an example which searches for all the email addresses, and changes them to keep the user (\1) but have yo-yo-dyne.com as the host.

```
str = 'purple alice@google.com, blah monkey bob@abc.com blah dishwasher'
## re.sub(pat, replacement, str) -- returns new string with all replacements,
## \1 is group(1), \2 group(2) in the replacement
print re.sub(r'([\w\.-]+)@([\w\.-]+)', r'\1@yo-yo-dyne.com', str)
## purple alice@yo-yo-dyne.com, blah monkey bob@yo-yo-dyne.com blah dishwasher
```