PYTHON NOTES

Table des matières

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

Les espaces

```
long_wined_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                          13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)

list_of_lists = [[1,2,3],
                 [4,5,6],
                 [7,8,9]]
```

Utilisation du \ ou des () pour coder sur plusieurs lignes

```
two_plus_three = 2 + \
                 3

three_plus_four = (3 +
                   4)
```

Les fonctions

Voici comment on définit une fonction

```
def double(x):
    """
    This is where you put an optional docstring that explains what the
    function does. For example, this function multiplies its input by 2.
    """
    return x * 2

print(type(double))

## <class 'function'>
```

En python, une fonction peut aussi avoir pour argument une autre fonction :

```
def apply_to_one(f):
    """Calls the function f with 1 as its argument"""
    return f(1)

my_double = double              # refers to the previously defined function
x = apply_to_one(my_double)     # equals 2
print(x)

## 2
```

On peut créer aussi des fonctions courtes ou fonctions anonymes appelées ***lambda expression***

```
lambda_exp = (lambda x: x + 4)
print(type(lambda_exp))

## <class 'function'>
```

```
y = apply_to_one(lambda_exp) # equals 5
print(y)

## 5
```

En général on déconseille l'assignation dans une variable d'une lambda expression

```
y = apply_to_one(lambda x: x) # equals 5
```

```
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#
      print(i)
```

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'))
```

Condition ternaire

Permet d'assigner des valeurs sans avoir à écrire la syntaxe classique d'une condition

```
x = 6
parity = "even" if x % 2 == 0 else "odd"
print(f"lorsque x vaut {x}, parity vaut {parity}")

## lorsque x vaut 6, parity vaut even

x = 7
parity = "even" if x % 2 == 0 else "odd"
print(f"lorsque x vaut {x}, parity vaut {parity}")

## lorsque x vaut 7, parity vaut odd
```

Les Booléens

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False

print(f"one_is_less_than_two vaut {one_is_less_than_two} et son type est {type(
one_is_less_than_two)}")

## one_is_less_than_two vaut True et son type est <class 'bool'>

print(f"one_is_less_than_two vaut {true_equals_false} et son type est {type(tru
e_equals_false)}")

## one_is_less_than_two vaut False et son type est <class 'bool'>
```

Les expressions évaluées à False

False
None
[] (an empty list)

{} (an empty dict)
""

set()
0
0.0

Les évaluations atypiques

```
def some_function_that_returns_a_string():
    """Function that return a String"""
    return "John Do"

s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""

print(first_char)

## J
```

Une façon plus courte d'écrire la condition ci-dessus est la suivante :

```
first_char = s and s[0]
print(type(True))

## <class 'bool'>

print(f"first_char vaut {first_char} et son type est {type(first_char)}")

## first_char vaut J et son type est <class 'str'>
```

On pourrait penser que dans l'exemple ci-dessus, **first_char** soit un **bool** or ce n'est pas le cas **first_char** est bel et bien un **str**, il y a bien une assignation qui a été faite. Autre exemple, du même type :

```
x = 3
safe_x = x or 0
print(safe_x)

## 3
```

Ce qui est équivalent à :

```
safe_x = x if x is not None else 0
```

Les fonctions all et any

Python a une fonction ***all***, qui prend un itérable et retourne Vrai précisément quand chaque élément est vrai, et une fonction ***any***, qui retourne Vrai quand au moins un élément est vrai :

```
all([True, 1, {3}])   # True, all are truthy

## True
```

```
all([True, 1, {}])    # False, {} is falsy

## False

any([True, 1, {}])    # True, True is truthy

## True

all([])               # True, no falsy elements in the list

## True

any([])               # False, no truthy elements in the list

## False
```

Exception TRY/CATCH

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

Automated Testing and Assert

En tant que spécialistes des données, nous allons écrire beaucoup de codes. Comment pouvons-nous être sûrs que notre code est correct ? D'une part, avec les types (dont nous parlerons bientôt), d'autre part, avec les tests automatisés.

On peut utiliser des déclarations d'assertion, qui feront que votre code produira une **AssertionError** si votre condition spécifiée n'est pas vraie :

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

Comme on peut le voir dans le second cas, on peut éventuellement ajouter un message à imprimer si l'affirmation échoue.

Il n'est pas particulièrement intéressant d'affirmer que 1 + 1 = 2. Ce qui est plus intéressant, c'est d'affirmer que les fonctions qu'on écrit font ce qu'on attend d'elles :

```
def smallest_item(xs):
    return min(xs)

assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

Une autre approche peu commune pourtant tout aussi intéressante est l'utilisation d'affirmation sur les inputs des fonctions.
Exemple:

```
def smallest_item(xs):
    assert xs, "empty list has no smallest item"
    return min(xs)
```

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
```

f-string

Une nouvelle fonctionnalité de Python 3.6 est la f-string, qui fournit un moyen simple de substituer des valeurs dans des chaînes de caractères. Par exemple, si nous avions le prénom et le nom de famille donnés séparément :

```
first_name = "Wis"
last_name = "Dom"
```

nous pourrions vouloir les combiner dans une variable **full_name**. Il existe plusieurs façons de construire une telle chaîne de **full_name** :

```
full_name1 = first_name + " " + last_name             # string addition
full_name2 = "{0} {1}".format(first_name, last_name)  # string.format
```

mais la façon ***f-string*** est beaucoup moins lourde :

```
full_name3 = f"{first_name} {last_name}"
print(full_name3)

## Wis Dom
```

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

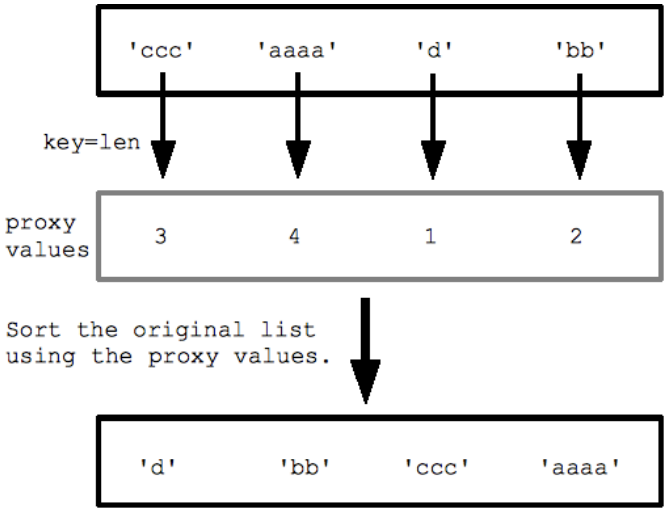
```
a = [5, 1, 4, 3]
print sorted(a)  ## [1, 3, 4, 5]
print a  ## [5, 1, 4, 3]

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
```

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 :

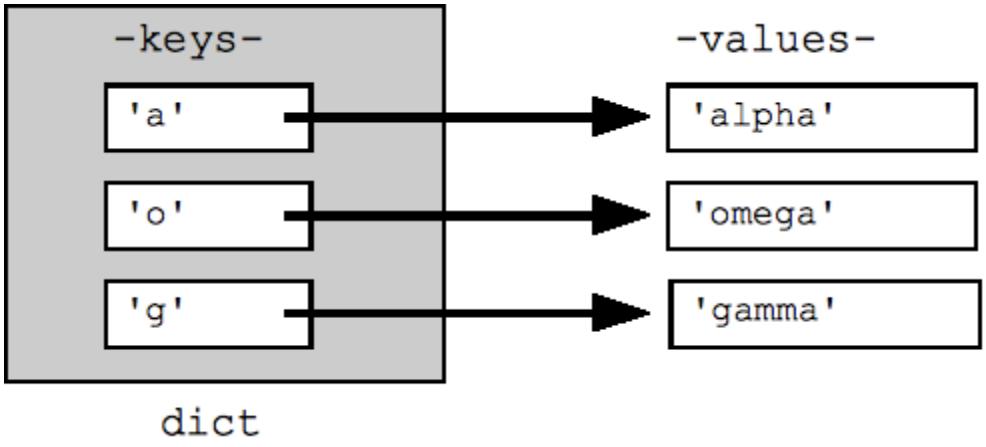
```
## Select values <= 2
nums = [2, 8, 1, 6]
small = [ n for n in nums if n <= 2 ]  ## [2, 1]

## 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']
```

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}
```

defaultdict

Utilisation de ***defaultdict*** :
Un ***defaultdict*** est comme un dictionnaire ordinaire, sauf que lorsque vous essayez de rechercher une clé qu'il ne contient pas, il lui ajoute d'abord une valeur en utilisant une fonction zero-argument que vous avez fournie lors de sa création. Pour pouvoir utiliser les defaultdicts, vous devez les importer du module ***collections*** :

```
from collections import defaultdict

l1 = ['cat','dog','parrot','snake','cat']

word_counts = defaultdict(int)          # int() produces 0
print("Avant comptage : ")

## Avant comptage :

print(word_counts)

## defaultdict(<class 'int'>, {})

for word in l1:
    word_counts[word] += 1

print("Après comptage : ")

## Après comptage :

print(word_counts)

## defaultdict(<class 'int'>, {'cat': 2, 'dog': 1, 'parrot': 1, 'snake': 1})
```

Il peuvent être aussi très utile avec des listes ou des dictionnaires ou nos propres fonctions

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # {"Joel" : {"City": Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0, 1]}
```

Ils seront utiles lorsque nous utiliserons des dictionnaires pour "collecter" les résultats par une clé quelconque et que nous ne voulons pas avoir à vérifier à chaque fois si la clé existe déjà.

Les Compteurs

Un compteur transforme une séquence de valeurs en un objet de type defaultdict(int) qui associe des clés de comptage :

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) {0: 2, 1: 1, 2: 1}
print(c)

## Counter({0: 2, 1: 1, 2: 1})
```

Cela permet de façon très simple de résoudre le problème du word_counts

```
# recall, document is a list of words
document = """
In the early days of artificial intelligence, the field rapidly tackled and sol
ved problems
that are intellectually difficult for human beings but relatively straightforwa
rd for
computers—problems that can be described by a list of formal, mathematical rule
s. The true
challenge to artificial intelligence proved to be solving the tasks that are ea
sy for people
to perform but hard for people to describe formally—problems that we solve intu
itively, that
feel automatic, like recognizingspoken words or faces in images.
""".replace('\n','').split(' ')
word_counts = Counter(document)
print(word_counts)

## Counter({'that': 5, 'for': 4, 'to': 4, 'the': 3, 'of': 2, 'artificial': 2, '
are': 2, 'but': 2, 'be': 2, 'people': 2, 'In': 1, 'early': 1, 'days': 1, 'intel
ligence,': 1, 'field': 1, 'rapidly': 1, 'tackled': 1, 'and': 1, 'solved': 1, 'p
roblems': 1, 'intellectually': 1, 'difficult': 1, 'human': 1, 'beings': 1, 'rel
atively': 1, 'straightforward': 1, 'computers—problems': 1, 'can': 1, 'describe
d': 1, 'by': 1, 'a': 1, 'list': 1, 'formal,': 1, 'mathematical': 1, 'rules.': 1
, 'The': 1, 'true': 1, 'challenge': 1, 'intelligence': 1, 'proved': 1, 'solving
': 1, 'tasks': 1, 'easy': 1, 'perform': 1, 'hard': 1, 'describe': 1, 'formally—
problems': 1, 'we': 1, 'solve': 1, 'intuitively,': 1, 'feel': 1, 'automatic,':
1, 'like': 1, 'recognizingspoken': 1, 'words': 1, 'or': 1, 'faces': 1, 'in': 1,
'images.': 1})
```

Une instance Counter dispose d'une méthode *most_common* qui est fréquemment utile :

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(5):
    print(word, count)

## that 5
## for 4
## to 4
## the 3
## of 2
```

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.

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
```

```
class Consumer:

    def __init__(self, w):
        "Initialize consumer with w dollars of wealth"
        self.wealth = w

    def earn(self, y):
        "The consumer earns y dollars"
        self.wealth += y

    def spend(self, x):
        "The consumer spends x dollars if feasible"
        new_wealth = self.wealth - x
        if new_wealth < 0:
            print("Insufficent funds")
        else:
            self.wealth = new wealth
```

Constructeur

Exemple classique d'une classe :

```
class CountingClicker:
    """A class can/should have a docstring, just like a function"""

    def __init__(self, count = 0):
        """Constructeur de la classe"""
        self.count = count

    def __repr__(self):
        """Représentation en chaînes de caractères de la classe"""
        return f"CountingClicker(count={self.count})"

    # Fonctions publiques
    def click(self, num_times = 1):
        """Click the clicker some number of times."""
        self.count += num_times

    def read(self):
        return self.count

    def reset(self):
        self.count = 0

clicker = CountingClicker()
assert clicker.read() == 0, "clicker should start with count 0"
clicker.click()
clicker.click()
assert clicker.read() == 2, "after two clicks, clicker should have count 2"
```

```
clicker.reset()
assert clicker.read() == 0, "after reset, clicker should be back to 0"
```

L'héritage

Voici un exemple où l'on créer une classe *NoResetClicker* qui hérite de la classe *CountingClicker*.

```
# A subclass inherits all the behavior of its parent class.
class NoResetClicker(CountingClicker):
    # This class has all the same methods as CountingClicker

    # Except that it has a reset method that does nothing.
    def reset(self):
        pass

clicker2 = NoResetClicker()
assert clicker2.read() == 0
clicker2.click()
assert clicker2.read() == 1
clicker2.reset()
assert clicker2.read() == 1, "reset shouldn't do anything"
```

La classe `Consumer` a pour attribut **wealth** et 3 méthodes (_), **wealth** est un attribut car à chaque création de client, chaque client à sa propre **wealth.**

In
```
c1 = Consumer(10)
c2 = Consumer(12)
c2.spend(4)
c2.wealth
```

Out 8

In
```
c1.wealth
```

Out 10

In fact each instance stores its data in a separate namespace dictionary

In
```
c1.__dict__
```

Out `{'wealth': 10}`

In
```
c2.__dict__
```

Out `{'wealth': 8}`

When we access or set attributes we're actually just modifying the dictionary maintained by the instance

Redéfinition de fonctions

```
class Foo:
  def __len__(self):
    return 17
  def __call__(self, x):
    return x + 42
f = Foo()
len(f) #17
f(8) #50 <==> f.__call__(8)
```

ITERATORS

Iterable

un objet est iterable si il peut être converti en itérateur avec la fonction iter()

Iterator

interface uniforme qui permet de parcourir les éléments d'une collection

Les listes sont des **itérables** (opposé d'un itérateur)

Ex :

```
x = {'foo', 'bar'}
type(x) #list
next(x) #TypeError : 'list object is not an iterator'

y = iter(x)
type(y) #list_iterator
next(y) #'foo'
next(y) #'bar'
next(y) #StopIteration Error ==> dans une boucle for cette error est le signal de la fin de la boucle
```

Les dictionnaires et les tuples sont des itérables

Names and Name Resolution

Les noms d'objets sont gérés par Python dans un dictionnaire.

```
def f(string):
  return string+'+'

g = f

id(g)=id(f) #True, car f et g ont le même nom objet
```

Namespace

Pour voir le contenu d'un espace noms (=dictionnaires)

```
import math

math.__dict__.items()      <==>   vars(math).items
dir(math)
```

Pour les fonctions imbriquées l'espace de nom est différent, Ici g se trouve dans un espace de nom local, alors que f se trouvera dans un espace de nom général.

```
def f():
  a = 2
  def g():
    b = 4
    print(a*b)
  g()
```

DECORATORS

Ex :

```
import numpy as np

def check_nonneg(func):
    def safe_function(x):
        assert x >= 0, "Argument must be nonnegative"
        return func(x)
    return safe_function

def f(x):
    return np.log(np.log(x))

def g(x):
    return np.sqrt(42 * x)

f = check_nonneg(f)
g = check_nonneg(g)
# Program continues with various calculations using f and g
```

Remplacer par

```
@check_nonneg
def f(x):
    return np.log(np.log(x))

@check_nonneg
def g(x):
    return np.sqrt(42 * x)
```

DESCRIPTORS

```
class Car:
  def __init__(self, miles=1000):
    self.__miles = miles
    self.__kms = miles*1.61

  def set_miles(self, value):
    self.__miles = value

  def get_miles(self):
    return self.__miles
```

Devient avec les descriptors.

```
class Car:
  def __init__(self, miles = 1000):
    self.__miles = miles
    self.__kms = miles*1.61

  @property
  def miles(self):
    return self.miles

  @property
  def kms(self):
    return self.kms

  @miles.setter
  def miles(self, value):
    self.__miles = value
    self.__kms = value*1.61
```

GENERATORS (fait partie de la famille des itérateurs)

Ex :

```
singular = ('dog', 'cat', 'bird')
plural = (string+'s' for string in singular)
type(plural) #generator

next(plural) #'dogs'
next(plural) #'cats'
next(plural) #'birds'
```

La fonction sum() peut être appelé sur un itérateur

Ex :

```
sum((x*x for x in range(10)))   #<==> sum(x*x for x in range(10))
```

Exemple de function générateur :

```
def f():
  yield 'start'
  yield 'middle'
  yield 'end'

type(f) #function
gen = f() #<== create a generator objects
gen #<generator object f at 0x7f957c23e840>
next(gen) #'start'
next(gen) #'middle'
next(gen) #'end'
next(gen) #StopIteration Error
```

Le premier appel **next(gen)** exécute le code du corps de la fonction **f()** jusqu'à rencontrer un **yield** statement puis retourne la valeur au caller de **next(gen)**.

Quand le block est fini, le générateur lance a StopIteration Error.

Autre exemple :

```
def g(x):
  while x<100:
    yield x
    x=x*x

g
gen = g(2)
type(gen) #generator
next(gen) #2
next(gen) #4
next(gen) #16
next(gen) #StopIteration error
```

avantage des générateurs:

```
n = 10000000
draws = [random.uniform(0,1)<0.5 for i in range(n)]
```

On crée ici 2 énormes listes qui usent beaucoup de mémoire et sont très lente à créer.
Pour éviter ce genre de problème, on peut utiliser des itérateurs particulièrement des **générateurs**.

```
def f(n):
  i=1
  while i<=n:
    yield random.uniform(0,1)<0.5
    i+=1

n = 10000000
draws = f(n)
draws #<generator object f at ...>
```

```
sum(draws) #5001892
```

En résumé, les itérables :

- **Permette d'éviter le besoin de créer d'immenses listes ou tuples**
- **Fournissent une interface uniforme pour itérer qui peut être utilisée de façon transparente dans un for**

En python tout est objet. Toute variable est passée par référence.

```
# Pour afficher le résultat de toutes les variables
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import sys

print ("Configuration de Python : ",sys.version)

## Configuration de Python :  3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.191
5 64 bit (AMD64)]
```

zip and Argument Unpacking

Souvent, on doit **zip** deux ou plusieurs itérateurs. La fonction **zip** transforme plusieurs itérables en un seul itérable de tuples de fonction correspondante :

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

# zip is lazy, so you have to do something like the following
[pair for pair in zip(list1, list2)]    # is [('a', 1), ('b', 2), ('c', 3)]

## [('a', 1), ('b', 2), ('c', 3)]
```

Si les listes sont de tailles différentes, la fonction **zip** stop des que la plus petite liste s'arrête.

On peut aussi **unzip** une liste, en utilisant l'astérix * comme suit :

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
print(f"pairs {pairs}")

## pairs [('a', 1), ('b', 2), ('c', 3)]

print(f"letters : {letters}")

## letters : ('a', 'b', 'c')

print(f"numbers : {numbers}")

## numbers : (1, 2, 3)
```

L'astérisque (*) engendre l' "*arguments unpacking*", qui utilisent les éléments des paires comme des arguments individuels à zipper.

Le résultat est le même que si vous aviez appelé :

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
print(f"letters : {letters}")

## letters : ('a', 'b', 'c')

print(f"numbers : {numbers}")

## numbers : (1, 2, 3)
```

args and kwargs

On a souvent besoin de créer des fonctions avec un nombre aribitraires d'arguments. On peut faire cela avec l' "*arguments unpacking*" de la façon suivante :

```
def magic(*args, **kwargs):
    print("unnamed args:", args)
    print("keyword args:", kwargs)

magic(1, 2, key="word", key2="word2")

## unnamed args: (1, 2)
## keyword args: {'key': 'word', 'key2': 'word2'}
```

Autrement dit, lorsque nous définissons une fonction comme celle-ci, args est un tuple de ses arguments non nommés et kwargs est un dict de ses arguments nommés. Cela fonctionne également dans l'autre sens, si on veut utiliser une liste (ou un n-uplet) et un dict pour fournir des arguments à une fonction :

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = {"z": 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 should be 6"

def f2(*args, **kwargs):
    return 3

def doubler_correct(f):
    """works no matter what kind of inputs f expects"""
    def g(*args, **kwargs):
        """whatever arguments g is supplied, pass them through to f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
assert g(1, 2) == 6, "doubler should work now"
```

Type Annotations

On a deux types de façons de déclarer une fonction :

```
def dot_product(x, y): ...

def dot_product(x: Vector, y: Vector) -> float: ...
```

Dans la première, on laisse Python deviner le type des variables, dans la seconde on le spécifie. Le module mypy permet de vérifier le type des variables des fonctions avant d'exécuter le code.

Une autre façon de spécifier les types est la suivante :

```
from typing import Union

def secretly_ugly_function(value, operation): ...
```

```
def ugly_function(value: int,
                  operation: Union[str, int, float, bool]) -> int:
    ...
```

Cette façon permet de simplifier la lecture et permet à l'éditeur de vous faire des suggestions sur les fonctions à utiliser. Voir le module typing.

How to Write Type Annotations

```
def total(xs: list) -> float:
    return sum(total)
```

Cette forme d'annotation n'est pas assez précise. Une façon plus précise est la suivante :

```
from typing import List  # note capital L

def total(xs: List[float]) -> float:
    return sum(total)
```

On peut aussi annotter le type de variables :

```
# This is how to type-annotate variables when you define them.
# But this is unnecessary; it's "obvious" x is an int.
x: int = 5
```

Malheureusement parfois ce n'est pas évident de savoir quel est le type de la variable comme dans les cas suivants :

```
values = []         # what's my type?
best_so_far = None  # what's my type?
```

Dans ces cas, on peut fournir des indices de la façon suivante :

```
from typing import Optional

values: List[int] = []
best_so_far: Optional[float] = None  # allowed to be either a float or None
```

Le module typing contient bien d'autres types :

```
# the type annotations in this snippet are all unnecessary
from typing import Dict, Iterable, Tuple

# keys are strings, values are ints
counts: Dict[str, int] = {'data': 1, 'science': 2}

lazy : bool = True

# lists and generators are both iterable
if lazy:
    evens: Iterable[int] = (x for x in range(10) if x % 2 == 0)
else:
    evens = [0, 2, 4, 6, 8]
```

```
# tuples specify a type for each element
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Enfin, puisque Python a des fonctions de premier ordre, nous avons besoin d'un type pour les représenter également. Voici un exemple assez artificiel :

```
from typing import Callable

# The type hint says that repeater is a function that takes
# two arguments, a string and an int, and returns a string.
def twice(repeater: Callable[[str, int], str], s: str) -> str:
    return repeater(s, 2)

def comma_repeater(s: str, n: int) -> str:
    n_copies = [s for _ in range(n)]
    return ', '.join(n_copies)

assert twice(comma_repeater, "type hints") == "type hints, type hints"
```

Comme les annotations de type ne sont que des objets Python, nous pouvons les attribuer à des variables pour qu'il soit plus facile de s'y référer :

```
Number = int
Numbers = List[Number]

def total(xs: Numbers) -> Number:
    return sum(xs)
```

Numpy

- `np.searchsorted([1,2,3,4,5], 3) #return first index of the element which is like >=3`
- `z = np.array([[1,2],[3,4]]) #initialising 2D array`
- `z.T #transpose of z`
- `np.asarray(z) #convert input as an array, we can precise the type of the elements`
- `z.argmax() #return the index of the max element in z`
- `z.cumsum() #return the cumulative sum of elements in z`
- `z.cumprod() #return the cumulative multiplication of elements in z`
- `z.var() #return the variance of z`
- `z.std() #return the standard deviation of z`
- `z.shape() #return a tuple with the dimension of z : (n_row, n_column)`
- `np.mean(z)`
- `np.sum(z)`
- `np.linspace(1,2,5) #array([1.,1.25,1.5,1.75,2.])`

- `tab1 = np.zeros((2,2)) #array([[0.,0.],[0.,0.]])`
- `tab2 = np.ones((2,2)) #array([[1,1],[1,1]])`

- `tab2 = tab2*10 #array([[10,10],[10,10]])`
- `tab1 = tab1 + np.array([[1,2],[3,4]]) #array([[1.,2.],[3.,4.]])`
- `tab1*tab2 #array([[10.,20.],[30.,40.]]) multiplication terme à terme`
- `tab1@tab2 #array([[30.,30.],[70.,70.,]]) multiplication matriciel`
- `tab1@(0,1) #array([2,4]) multiplication matriciel`

- `z = np.random.randn(4) #array([-0.03131747,  0.61709198, -0.21803826, 0.09453925])`
- `np.where(z>0.2, 1 ,0) #Insert 1 if x > 0 true, otherwise 0 => array([0, 1, 1, 0])`
- `def f(x): return 1 if x > 0 else 0`

  `f = np.vectorize(f)`
  `f(z) #array([0, 1, 1, 0]) We get the same result as later(sometime vectorize is slower than where)`

Comparaison

- `z = np.array([2, 3])`
- `y = np.array([2, 3])`

- `z == y #array([True, True])`

- `y[0] = 5`

- `z==y #array([False,True])`
- `z!=y #array([True,False])`
- `z>1 #array([True, True])`

- `tab1 = np.array([2,3,6])`
- `tab2 = tab1`
- `tab1[0]=5`

- `tab1==tab2 #tab1 et tab2 pointe vers la même adresse mémoire =>array([True, True, True])`

- `tab = np.linspace(0,10,5) #array([0., 2.5, 5., 7.5, 10])`

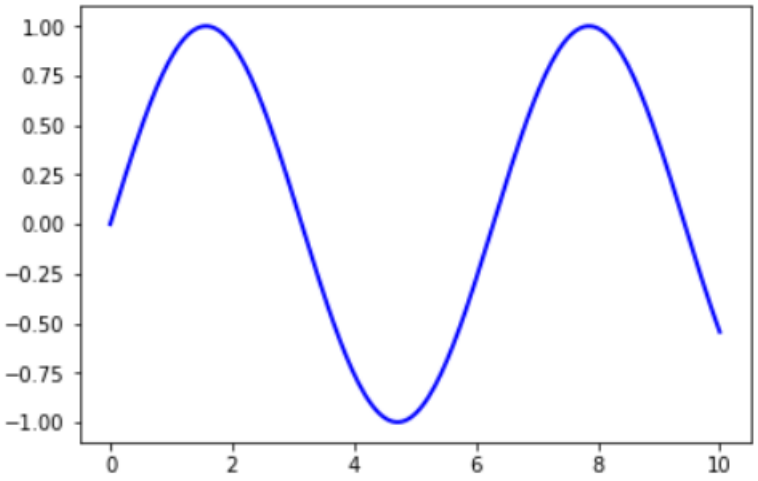
- `r = tab>3 #array([False, False, True, True, True])`
- `tab[r] #array([5., 7., 10])`
- `tab[tab>3] #array([5., 7., 10])`

Subpackages

- `z = np.random.randn(10000)  #Generate standard normals`
- `y = np.random.binomial(10, 0.5, size=1000) #1,000 draws from Bin(10, 0.5)`
- `y.mean()`

- `A = np.array([[1, 2], [3, 4]])`
- `np.linalg.det(A) #Compute the determinant of the matrix => -2.000000000000004`
- `np.linalg.inv(A) #Compute the inverse => array([[-2., 1.], [1.5, -0.5]])`

- `fig, ax = plt.subplots()`
- `ax.plot(x, y, 'b-', linewidth=2)`
- `plt.show()`

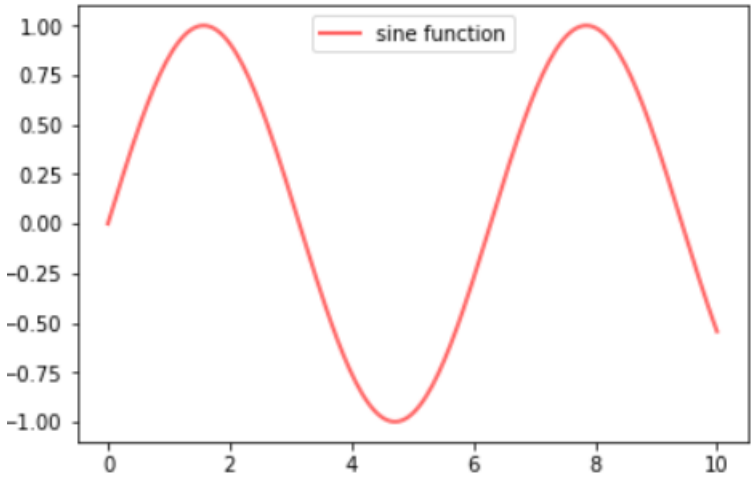


Here the call *fig, ax = plt.subplots()* returns a pair, where

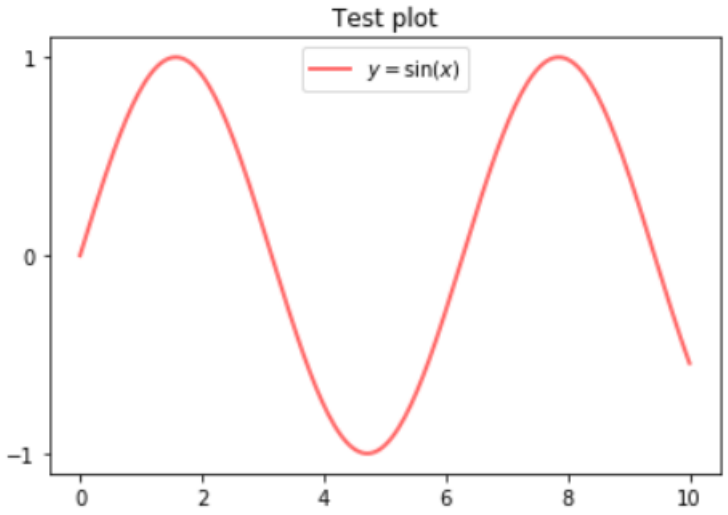
- *fig* is a Figure instance—like a blank canvas
- *ax* is an AxesSubplot instance—think of a frame for plotting in

The plot() function is actually a method of ax
While there's a bit more typing, the more explicit use of objects gives us better control

```
fig, ax = plt.subplots()
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='upper center')
plt.show()
```



```
fig, ax = plt.subplots()
ax.plot(x, y, 'r-', linewidth=2, label='$y=\sin(x)$', alpha=0.6)
ax.legend(loc='upper center')
ax.set_yticks([-1, 0, 1])
ax.set_title('Test plot')
plt.show()
```

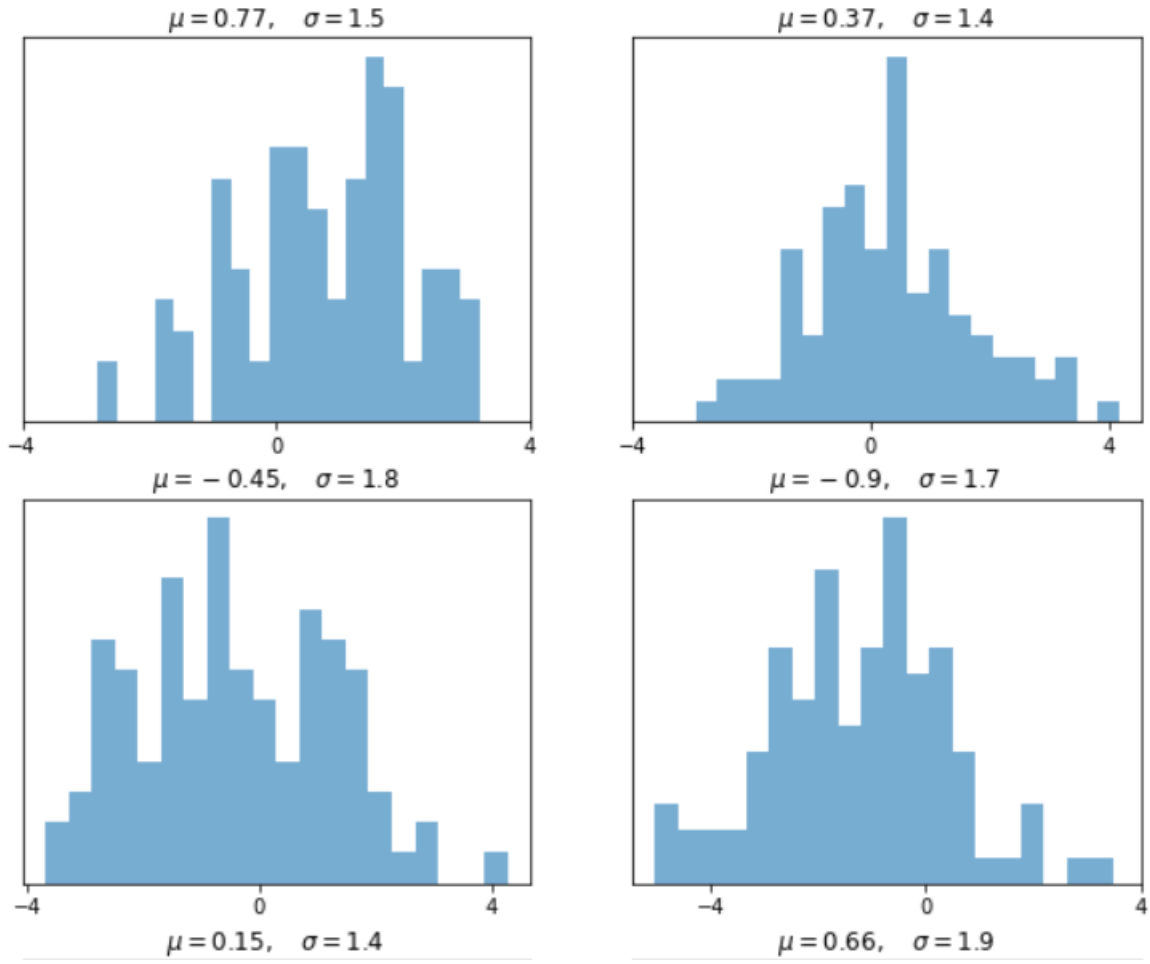


Multiple Plots on One Axis

```
from scipy.stats import norm
from random import uniform

fig, ax = plt.subplots()
x = np.linspace(-4, 4, 150)
for i in range(3):
    m, s = uniform(-1, 1), uniform(1, 2)
    y = norm.pdf(x, loc=m, scale=s)
    current_label = f'$\mu = {m:.2}$'
    ax.plot(x, y, linewidth=2, alpha=0.6, label=current_label)
ax.legend()
plt.show()
```

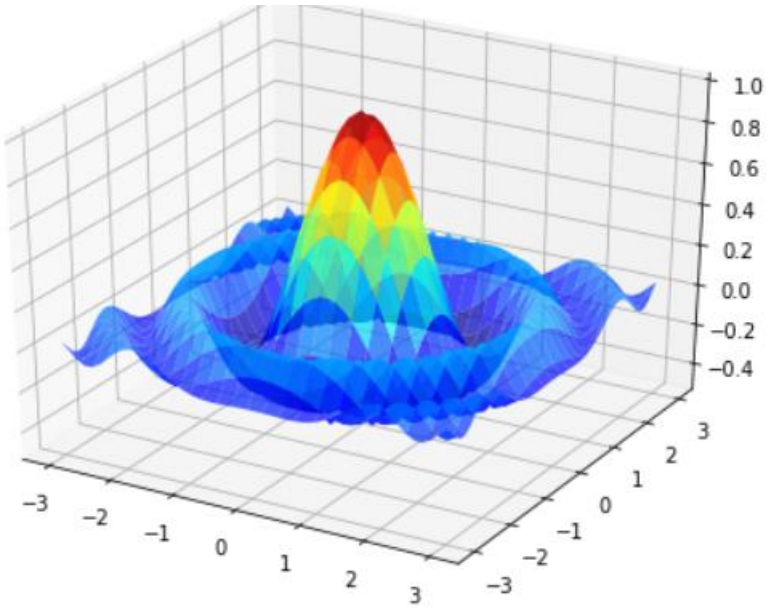

```
num_rows, num_cols = 3, 2
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 12))
for i in range(num_rows):
    for j in range(num_cols):
        m, s = uniform(-1, 1), uniform(1, 2)
        x = norm.rvs(loc=m, scale=s, size=100)
        axes[i, j].hist(x, alpha=0.6, bins=20)
        t = f'$\mu = {m:.2}, \quad \sigma = {s:.2}$'
        axes[i, j].set(title=t, xticks=[-4, 0, 4], yticks=[])
plt.show()
```

Multiple Subplots


$\mu = 0.77, \quad \sigma = 1.5$
$\mu = 0.37, \quad \sigma = 1.4$
$\mu = -0.45, \quad \sigma = 1.8$
$\mu = -0.9, \quad \sigma = 1.7$
$\mu = 0.15, \quad \sigma = 1.4$
$\mu = 0.66, \quad \sigma = 1.9$
−4
0
4

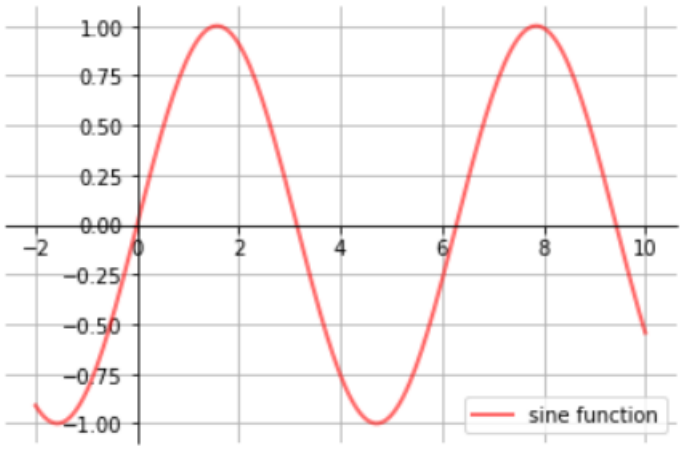
3D Plots

```
def subplots():
    "Custom subplots with axes throught the origin"
    fig, ax = plt.subplots()

    # Set the axes through the origin
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    ax.grid()
    return fig, ax


fig, ax = subplots()  # Call the local version, not plt.subplots()
x = np.linspace(-2, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='lower right')
plt.show()
```



Customizing Function

```
def subplots():
    "Custom subplots with axes throught the origin"
    fig, ax = plt.subplots()

    # Set the axes through the origin
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    ax.grid()
    return fig, ax


fig, ax = subplots()  # Call the local version, not plt.subplots()
x = np.linspace(-2, 10, 200)
y = np.sin(x)
ax.plot(x, y, 'r-', linewidth=2, label='sine function', alpha=0.6)
ax.legend(loc='lower right')
plt.show()
```



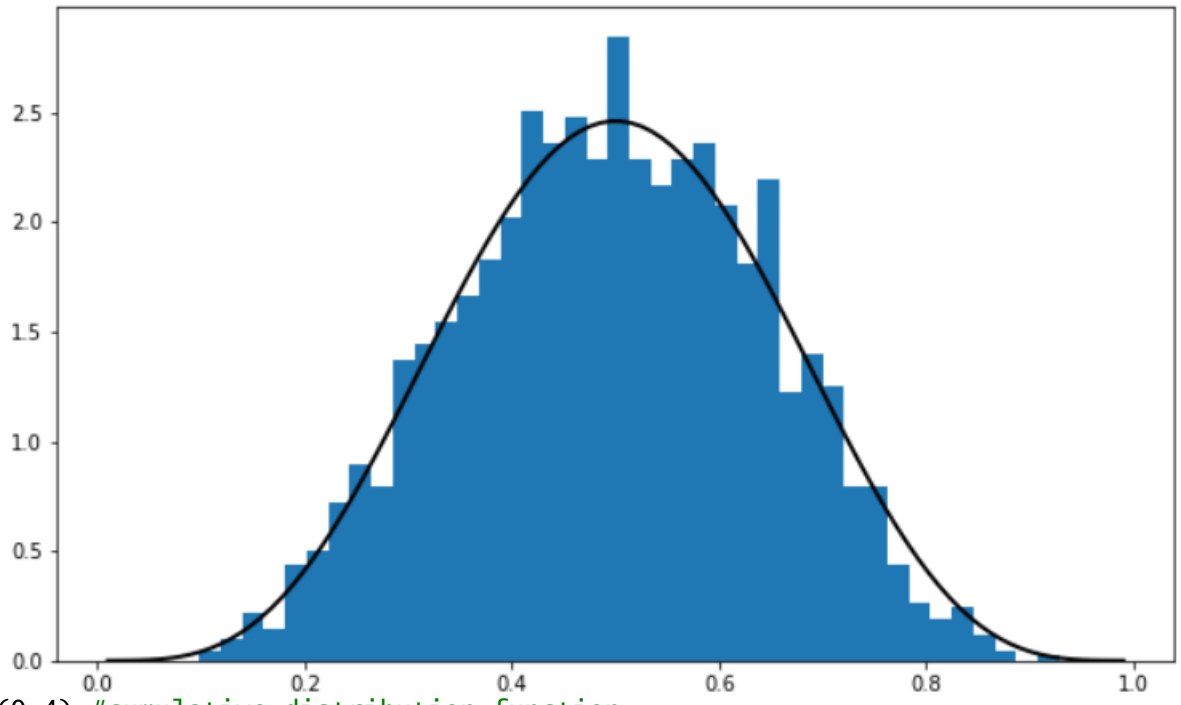
The custom subplots function

1. **calls the standard plt.subplots function internally to generate the fig, ax pair,**
2. **makes the desired customizations to ax, and**
3. **passes the fig, ax pair back to the calling code**

scipy.stats

q est un objet rv_frozen

```
from scipy.stats import beta
import matplotlib.pyplot as plt
%matplotlib inline

q = beta(5, 5)       # Beta(a, b), with a = b = 5
obs = q.rvs(2000)    # 2000 observations
grid = np.linspace(0.01, 0.99, 100)

fig, ax = plt.subplots(figsize=(10, 6))
ax.hist(obs, bins=40, density=True)
ax.plot(grid, q.pdf(grid), 'k-', linewidth=2)
plt.show()
```




```
q.cdf(0.4) #cumulative distribution function
q.pdf(0.4) #density function
q.ppf(0.8) #quantile function(Inverse of cdf)
q.mean()
```

```
identifier = scipy.stats.distribution_name(shape_parameters) #distribution_name nom de la distribution dans le module scipy.stats

autre façon :
identifier = scipy.stats.distribution_name(shape_parameters, loc = c, scale = d)
#transform X into Y = c + dX
```

Régression Linéaire

```
from scipy.stats import linregress
x = np.random.randn(200)
y = 2*x+0.1*np.random.randn(200)
gradient, intercept, r.value, p.value, std_err = linregress(x, y)
gradient, intecept
```

scipy.optimize

Racines et Points Fixes

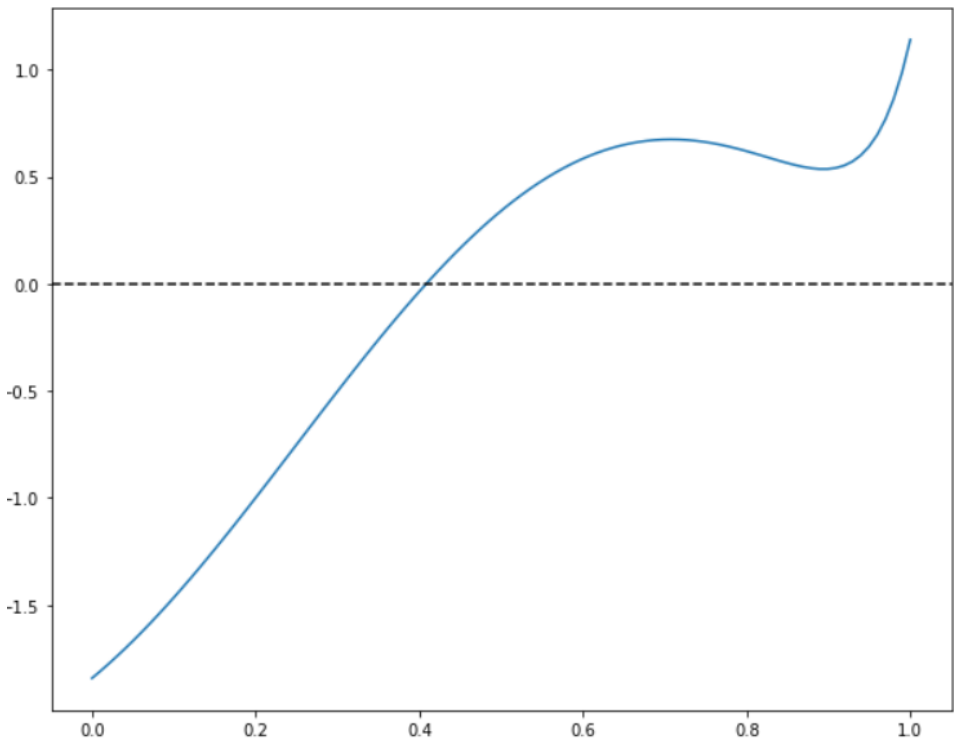
A *root* of a real function f on $[a, b]$ is an $x \in [a, b]$ such that $f(x) = 0$

For example, if we plot the function

$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1$$

with $x \in [0, 1]$ we get

```
f = lambda x: np.sin(4 * (x - 1/4)) + x + x**20 - 1
x = np.linspace(0, 1, 100)

plt.figure(figsize=(10, 8))
plt.plot(x, f(x))
plt.axhline(ls='--', c='k')
plt.show()
```




```
bisect(f, 0, 1) #0.48 sur [0,1] Bisection Method
newton(f, 0.2) #0.48 avec condition initial x=0.2 Newton-Raphson Method
brentq(f, 0, 1) #0.48 sur [0,1] Hybrid Method
```

Recherche de racines multiples

Utiliser **scipy.optimize.fsolve**

Points fixes

```
from scipy.optimize import fixed_point
fixed_point(lambda x : x**2, 10.0) #10 est la valeur qu'on pense pour le point fixe
```

Optimization

```
from scipy.optimize import fminbound
fminbound(lambda x : x**2, -1, 2)  #Recherche le minimum sur [-1,2]
```

Multivariate Optimization

Multivariate local optimizers contiennent:

- minimize
- fmin
- fmin_powell
- fmin_cg
- fmin_bfgs
- fmin_ncg

Constrained multivariate local optimizers contiennent:

- fmin_l_bfgs_b
- fmin_tnc
- fmin_cobyla

scipy.integrate

Integration

```
from scipy.integrate import quad
integral, error = quad(lambda x : x**2, 0, 1)
integral #0.333 Clenshaw-Curtis quadrature better use fixed_quad
```

```
from numba import jit
def qm(x0, n):
      x = np.empty(n+1)
      x[0] = x0
      for t in range(n):
            x[t+1] = 4*x[t]*(1-x[t])
      return x

qm_numba = jit(qm)
qm_numba(0.1, int(10**5)) #1 seconde + rapide que qm
```

Decorator Notation

Si on ne veut pas donner de nom à part pour les fonctions qui utilisent numba, on peut utiliser un décorateur.

```
@jit  #décorateur
def qm(x0, n) :
      x = np.empty(n+1)
      x[0] = x0
      for t in range(n):
            x[t+1] = 4*x[t]*(1-x[t])
      return x
```

Le code ci-dessus ⇔ qm = jit(qm)

Universal Functions(ufunc)

Many functions provided by Numpy are so called universal functions also called ufuncs.
This means that they:

- maps scalars into scalars, as expected
- maps arrays into arrays, acting elements-wise

Numba peut aussi être utilisé pour créer ses propres fonctions universel (ufuncs) avec le décorateur @vectorize

```
from numba import vectorize

@vectorize
def f_vec(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

grid = np.linspace(-3, 3, 1000)
x, y = np.meshgrid(grid, grid)

np.max(f_vec(x, y))  # Run once to compile

qe.tic() #quantecon module for mesuring time
np.max(f_vec(x, y))
qe.toc() #quantecon module for mesuring time
```

On gagne en rapidité par rapport à la ufunc de Numpy

Explication :
La rapidité est dû à l'usage moindre de la mémoire.

Ex : quand Numpy calcul np.cos(x2+y**2), il crée d'abord les tableaux intermédiaires x**2 et y**2, puis il crée le tableau np.cos(x**2+y**2)**

Dans la version utilisant NUMBA, l'opération entière est réduit a un simple processus de vectorization et aucun tableau intermédiaire n'est créé.

On peut encore gagner en rapidité, en utilisant la parallèlisation automatique en spécifiant `target='parallel`, mieux vaut alors préciser le type des variables d'entrées et de sorties.

```
@vectorize('float64(float64, float64)', target='parallel')
def f_vec(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

np.max(f_vec(x, y))  # Run once to compile

qe.tic()
np.max(f_vec(x, y))
qe.toc()
```

Cython

Cython apporte une approche pour générer du code rapidement compilable qui peut être utilisé sur Python.
Les développeurs ajoutent directement les types des variables dans leur code.

Cython permet aussi de traduire le code en code C et C++ optimisé, et de créer des extensions compatibles dans ces langages.

Ex :

`%load_ext Cython` ➔ run Cython in Jupyter notebook

`%%cython` ➔ indique à Jupyter qu'on écrit en Cython

```
def geo_prog_cython(double alpha, int n):
    cdef double current = 1.0
    cdef double sum = current
    cdef int i
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum
```

`cdef` mot clé indiquant la déclaration d'une variable avec son type.

Numba est plus rapide que Cython, car dans le code Cython, on utilise des structures de variables Python, pour accélérer le code mieux vaut utiliser des structures variables Cython.
Une astuce consiste à utiliser les structures de variables Cython et de les convertir en structures de variables Python.

```
%%cython
import numpy as np
from numpy cimport float_t

def qm_cython(double x0, int n):
    cdef int t
    x_np_array = np.zeros(n+1, dtype=float)
    cdef float_t [:] x = x_np_array
    x[0] = x0
    for t in range(n):
        x[t+1] = 4.0 * x[t] * (1 - x[t])
    return np.asarray(x)
```

Here

- `cimport` pulls in some compile-time information from NumPy
- `cdef float_t [:] x = x_np_array` creates a memoryview on the NumPy array `x_np_array`
- the return statement uses `np.asarray(x)` to convert the memoryview back to a NumPy array

Joblib : Librairie pour créer des caches et parallèliser

In
```
from joblib import Memory

memory = Memory(cachedir='./joblib_cache')

@memory.cache
def qm(x0, n):
    x = np.empty(n+1)
    x[0] = x0
    for t in range(n):
        x[t+1] = 4 * x[t] * (1 - x[t])
    return np.mean(x < 0.1)
```

Out
```
/home/quantecon/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
DeprecationWarning: The 'cachedir' parameter has been deprecated in version 0.12 and will be
removed in version 0.14.
You provided "cachedir='./joblib_cache'", use "location='./joblib_cache'" instead.
  This is separate from the ipykernel package so we can avoid doing imports until
```

We are using joblib to cache the result of calling qm at a given set of parameters

With the argument cachedir='./joblib_cache', any call to this function results in both the input values and output values being stored a subdirectory joblib_cache of the present working directory

The first time we call the function with a given set of parameters we see some extra output that notes information being cached

In
```
qe.util.tic()
n = int(1e7)
qm(0.2, n)
qe.util.toc()
```

Out
```
________________________________________________________________________________
[Memory] Calling __main__--home-quantecon-repos-collab-quantecon.build.lectures-
_build_jupyter-py-__ipython-input__.qm...
qm(0.2, 10000000)
_____________________________________________________________qm - 6.8s, 0.1min
TOC: Elapsed: 0:00:6.78

6.789849042892456
```

The next time we call the function with the same set of parameters, the result is returned almost instantaneously

In
```
qe.util.tic()
n = int(1e7)
qm(0.2, n)
qe.util.toc()
```

Out
```
TOC: Elapsed: 0:00:0.00

0.0008275508880615234
```