Programmation pour la Data-Science:

Python

Léo Beaucourt

Contenu du cours

- CM: Présentation de Python, introduction aux notions de base
- TP1: Variables, opérations de base, fonctions et numpy
- TP2: Manipulation de données avec pandas et visualisation I
- TP3: Git, classes et modules I
- TP4: Git, classes et modules II
- TP5: Manipulation de données avec pandas et visualisation II
- Examen

Calendrier & intervenants

CM (2h)	03/10/2018	Léo Beaucourt
TP1 (4h)	03/10/2018	Léo Beaucourt
TP2 (4h)	05/10/2018	Fabien Taghon
TP3 (4h)	05/11/2018	Antoine Dumas
TP4 (2h)	09/11/2018	Antoine Dumas
TP5 (4h)	12/12/2018	Fabien Taghon
Exam (4h)	07/02/2019	Antoine Dumas & Léo Beaucourt

Antoine Dumas: *dumas@phimeca.com*
Fabien Taghon: *taghon@phimeca.com*
Léo Beaucourt: *lbeaucourt@agaetis.fr*

Teasing et sources

- Utilisation en notebook / classique
- Beaucoup de possibilités!
- Sources d'informations (tutoriels):
  - Python: docs.python.org/3/tutorial/index.html
  - Tutoriel python par scipy: www.scipy-lectures.org/intro/index.html
  - Numpy: docs.scipy.org/doc/numpy/user/quickstart.html
  - Pandas: pandas.pydata.org/pandas-docs/stable/
- Vos meilleurs amis:
  - Google (ou bien Qwant si vous préférez!)
  - Stackoverflow

Généralités sur Python

- Langage de programmation créé en 1990
  par *Guido van Rossum* $\Rightarrow$
- Interprété (et non compilé comme le C)
  $\rightarrow$ Interactif (ex: notebooks)
- Gratuit et Open-source (www.python.org)
- Multi plateforme (Windows, **Linux**, ~~IOS~~, ...)

Généralités sur Python 2

```python
name = "Python"

if name == "Python":
    print("Python is cool!")
elif name == "Léo":
    print("Hello Léo, Nice to see you!")
elif name == "":
    print("Hello World!")
else:
    print("Hello {}".format(name))
```

```
Python is cool!
```

- Lisibilité et non verbeux
- Communauté: packages
- Multiusage: web, science, ...
- Interface avec d'autres langages

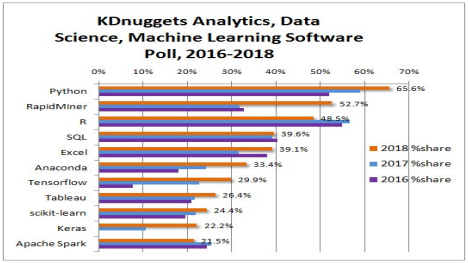
- Python utilise un *typing* dynamique
- Paradigmes de programmation et Python:
  - Orienté Object (POO)
  - Fonctionnel **ET** procédural
  - Impérative

Python en Data Science

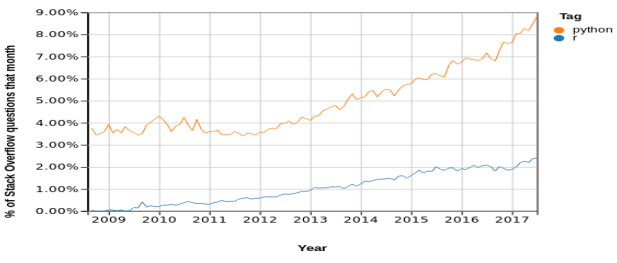
- Historiquement un langage de développeur → traitement des données
- Performant pour le calcul matriciel (avec numpy)
- Multi plateforme et non compilé (interactif → Jupyter)
- Multi usage: Exploration, prototype et industrialisation
- Beaucoup de packages utiles en Data Science:
  - Calcul numérique: **numpy**
  - Statistique: **openturns**
  - Manipulation des données: **pandas**
  - Visualisation: **matplotlib**, **seaborn**
  - Machine learning: **scikit-learn**
  - Deep learning: **tensorflow**, **pytorch**
  - …

Python versus R

- Python ⇒ Approche Machine Learning
- R ⇒ Approche Statistique
- Python is winning the game:



KDnuggets



stackoverflow

Types numériques

Entiers (*integer*)	`<type 'int'>`	a = 2 - 3
Flottants (*float*)	`<type 'float'>`	b = 3.1456
Complexes (*complex*)	`<type 'complex'>`	c = 1.5 + 0.4j
Booléens (*boolean*)	`<type 'bool'>`	d = (8 < 2.5)

- Conversion de types (*casting*): float(2) = 2.0
  - casting *float* $\rightarrow$ *int* = troncature: int(2.8) = 2
  - Utiliser l'arrondi: round(2.8) = 3.0

Types numériques: Opérations de base

- Addition, soustraction, multiplication, division: +, −, *, /
- Modulo: %: 8 % 3 = 2 (fonctionne aussi avec des *floats*)
- Division d'entiers **Attention:** Différence entre *python 2* et *python 3*

python 2	python 3	
3 / 2 = 1	3 / 2 = 1.5	
3 / 2. = 1.5	3 / 2. = 1.5	Utiliser des *floats*
3 / float(2) = 1.5	3 / float(2) = 1.5	ou le *casting*
3 // 2 = 1	3 // 2 = 1	Division d'entiers explicite
3.0 // 2 = 1.0	3 // 2.5 = 1.0	

Conteneurs

- 3 types majeurs de conteneurs:
  - Liste (*List*): `[2.5,'Price','exchange', 5]`
  - Chaîne de caractères (*String*): `'Hello World!'`
  - Dictionnaires (*Dictionnary*): `{'a':5,'b':9.2,'h':'ht'}`
- Autres types:
  - Tuples: `t = (9.2, 'h', 'ht')`
    - Les valeurs internes d'un *tuple* sont non modifiables
  - Sets: `s = set(('a', 'b', 'c', 'd'))`

Listes

- `<class 'list'>`
- Collection **ordonnée** d'objets
- Les types peuvent être différents
- Indexés: `liste[i]`
- Indices commencent à 0
- `l[d:f]`:
  - retourne `f-d` éléments
  - depuis `l[d]` à `l[f-1]`

```
l = ['this', 'is', 45, 5.2, True]
print(type(l))
l[2]
```

```
<class 'list'>
45
```

```
l[-4]
```

```
'is'
```

```
l[2:5]
```

```
[45, 5.2, True]
```

```
l[3:]
```

```
[5.2, True]
```

```
l[:4]
```

```
['this', 'is', 45, 5.2]
```

```
l[::2]
```

```
['this', 45, True]
```

Listes: fonctions et méthodes

- python propose beaucoup de fonctions pour travailler avec les listes
- Voir ici pour plus d'informations
- Ajouter et retirer des éléments: `append()`, `pop()`, `extend()`
- Renverser l'ordre: `reverse()`
- Concaténer: `l1 + l2`, `2 * l1`
- trier: `sorted()`, `sort()`

```
l.append(8)
l
```

```
['this', 'is', 45, 5.2, True, 8]
```

```
l.pop()
```

```
8
```

```
l
```

```
['this', 'is', 45, 5.2, True]
```

```
l.extend([9])
l
```

```
['this', 'is', 45, 5.2, True, 9]
```

```
l2 = list(l)
l2.reverse()
l2
```

```
[9, True, 5.2, 45, 'is', 'this']
```

```
l = l + l2
l
```

```
['this', 'is', 45, 5.2, True, 9, 9, True, 5.2, 45, 'is', 'this']
```

```
sorted(l[2:-2])
```

```
[True, True, 5.2, 5.2, 9, 9, 45, 45]
```

```
l2 = l[2:-2]
l2.sort()
l2
```

```
[True, True, 5.2, 5.2, 9, 9, 45, 45]
```

String

- `<class 'str'>`
- Similaire aux listes (index)
- Objets immuables ($\neq$ list)

```
# Simple quote
s = 'Hello World!'

# Double quote (tu use ' in sentences)
s = "Hello, I'm your python instructor."

# Triple quote (to allow \n line break)
s = '''Hello world,
nice to meet you!'''

s = """Hello,
I'm your python instructor.

We will see how to use strings."""

print(s)
```

```
Hello,
I'm your python instructor.

We will see how to use strings.
```

```
s = "Hello, I'm your python instructor."
s[7:15] + s[22:]
```

```
"I'm your instructor."
```

```
s[9] = 'g'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-82-fd823d94b55d> in <module>()
----> 1 s[9] = 'g'

TypeError: 'str' object does not support item assignment
```

```
s.replace('your','not your')
```

```
"Hello, I'm not your python instructor."
```

```
a = 2
b = 4
s = """Today, we have a {}-hour class this morning,
then {} hours this afternoon.""".format(a,b)
print(s)
```

```
Today, we have a 2-hour class this morning,
then 4 hours this afternoon.
```

Dictionnaires

- `<class 'dict'>`
- Ensemble de couples `{clé: valeur}`
- Conteneur **non ordonné** ($\neq$ list, string): **dict[i] interdit**
- Un dictionnaire peut avoir des clés et/ou des valeurs de types différents

```
d = {'a':5, 'g':8.89, 9: ['a', 2, 3], 7: 'hello'}
print(d['a'])
print(d[9])
```

```
5
['a', 2, 3]
```

```
d.keys()
```

```
dict_keys([9, 'g', 'a', 7])
```

```
d.values()
```

```
dict_values([['a', 2, 3], 8.89, 5, 'hello'])
```

Structures conditionnelles: if/elif/else

- Condition: booléen (**True** ou **False**)
- Comparaison: `==` `<` `<=` `>` `>=` `<` `!=`
- Tester l'identité: `a is b` (`a is not b`)
- Tester la présence: `a in col` (`a not in col`)
- Combiner des conditions: **and** (&) **or** (|)

```
1 == 1.
```

```
True
```

```
1 is 1.
```

```
False
```

```
if True:
    print("This is true")
```

```
This is true
```

```
a = 6

if type(a) is not int:
    print('{} is not an integer'.format(a))
elif a == 0:
    print("Input value is 0")
elif(a < 0) | (a > 10):
    print("{} is not in [0,10] range".format(a))
elif a in [2, 4, 6, 8, 10]:
    print("{} is an even value".format(a))
else:
    print("{} is an odd value".format(a))
```

```
6 is an even value
```

Structures itératives: for

- Répète un bloc d'instructions un nombre donné de fois

```
for i in range(3):
    print(i)
```

```
0
1
2
```

```
for i in range(2,5):
    print(i)
```

```
2
3
4
```

```
for i in range(0,5,2):
    print(i)
```

```
0
2
4
```

- On peut aussi itérer sur tous les éléments d'une collection

```
colors = ['blue', 'red', 'green',
          'yellow', 'purple']
for color in colors:
    print(color)
```

```
blue
red
green
yellow
purple
```

```
for c in 'Hello!':
    print(c)
```

```
H
e
l
l
o
!
```

```
dic = {'a':1, 'b':2, 'c':3}
for e in dic:
    print(e)
```

```
c
b
a
```

Structures itératives: while

- Tant que la condition est vraie: exécuter le bloc d'instructions
- **break**: Arrête la boucle
- **continue**: n'exécute pas les instructions restantes

```
i = 0
while i < 10:
    i = i + 1
print(i)
```

```
10
```

```
i = 1
while i > 0:
    if i >= 10:
        break
    i = i + 1
print(i)
```

```
10
```

```
i = -5
while i < 5:
    if i == 0:
        i = i + 1
        continue
    print(1/i)
    i = i + 1
```

```
-0.2
-0.25
-0.3333333333333333
-0.5
-1.0
1.0
0.5
0.3333333333333333
0.25
```

Structures itératives: bonus

- Afficher tous les mots d'une phrase
- Utiliser un compteur
- Itérer sur un dictionnaire
- Compréhension de liste

```
dic = {'a': 1, 'b': 2.2, 'c': 23.2}
for key, val in dic.items():
    print("key {} has a value {}".format(key, val))
```

```
key a has a value 1
key c has a value 23.2
key b has a value 2.2
```

```
[x**2 for x in range(5)]
```

```
[0, 1, 4, 9, 16]
```

```
sentence = "Python is a wonderful\
           tool for data science!"
for word in sentence.split():
    print(word)
```

```
Python
is
a
wonderful
tool
for
data
science!
```

```
for index, val in enumerate(word):
    print("{}: {}".format(index, val))
```

```
0: s
1: c
2: i
3: e
4: n
5: c
6: e
7: !
```

Les fonctions

- Mathématique: $y = f(x)$
- Informatique: `result = fonctionName(parameters)`

```
def awesomeFunction():
    print("Legendary!")

awesomeFunction()
```

```
Legendary!
```

```
def mySquare(x):
    return x**2

mySquare(5)
```

```
25
```

- **def**: mot clé indiquant le début d'une fonction
- **return**: permet à la fonction de renvoyer un objet
  - **return locals**: renvoie les variables définies dans la fonction
- Une fonction renvoie **None** par défaut
- Une fonction est **déclarée** puis **appelée**

Les fonctions: paramètres

- Obligatoires: doivent être renseignés lors de l'appel
- Optionnels: la fonction est définie avec des valeurs par défaut

```
def mandatoryParam(param):
    print(param)

mandatoryParam('Hi!')
mandatoryParam()
```

```
Hi!

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-11524a2f07de> in <module>()
      3
      4 mandatoryParam('Hi!')
----> 5 mandatoryParam()

TypeError: mandatoryParam() missing 1 required positional argument: 'param'
```

```
def optionalParam(param = 'Hello'):
    print(param)

optionalParam('Great')
optionalParam()
```

```
Great
Hello
```

Les fonctions: paramètres

- Les paramètres optionnels sont évalués lors de la déclaration
- Attention avec des objets variables qui sont modifiés dans la fonction

```
defArg = 20

def myFct(arg = defArg):
    return arg ** 2

print('{} -> {}'.format(defArg, myFct()))

defArg = 1e9
print('{} -> {}'.format(defArg, myFct()))
```

```
20 -> 400
1000000000.0 -> 400
```

```
def modDict(dic = {'a':1, 'b': 2}):
    for key in dic.keys():
        dic[key] += 1
    print(dic)

modDict()
modDict()
modDict()
```

```
{'b': 3, 'a': 2}
{'b': 4, 'a': 3}
{'b': 5, 'a': 4}
```

- l'ordre des paramètres optionnels n'est pas important
- Déclaration: paramètres obligatoires **avant** optionnels

```
def multiArgFct(arg1, arg2 = 1, arg3 = 4, arg4 = -8):
    print(arg1 + arg2 + arg3 + arg4)

multiArgFct(8)
multiArgFct(8, arg4 = 7, arg2 = -0.0123)
```

```
5
18.9877
```

Les variables en Python

- Définies **par références**
- Peuvent être **immuables** (*int*, *string*, ...) ou pas (*dict*, *list*)

```
x = 20
y = x
print('x = {}, y = {}'.format(x,y))
y += 2
print('x = {}, y = {}'.format(x,y))
```

```
x = 20, y = 20
x = 20, y = 22
```

```
x = [2]
y = x
print('x = {}, y = {}'.format(x,y))
y.append(5)
print('x = {}, y = {}'.format(x,y))
y = [6]
print('x = {}, y = {}'.format(x,y))
```

```
x = [2], y = [2]
x = [2, 5], y = [2, 5]
x = [2, 5], y = [6]
```

```
x = {'a': 5}
y = x
print('x = {}, y = {}'.format(x,y))
y['b'] = 9
print('x = {}, y = {}'.format(x,y))
```

```
x = {'a': 5}, y = {'a': 5}
x = {'b': 9, 'a': 5}, y = {'b': 9, 'a': 5}
```

```
x = {'a': 5}
y = x.copy()
print('x = {}, y = {}'.format(x,y))
y['b'] = 9
print('x = {}, y = {}'.format(x,y))
```

```
x = {'a': 5}, y = {'a': 5}
x = {'a': 5}, y = {'b': 9, 'a': 5}
```

- Les objets de type **dict** ou **list** ont une fonction `.copy()`

Les fonctions: passer les paramètres par valeurs

- Certains langages font la distinction entre:
  - *Passage par valeur*: la valeur de la variable est envoyé dans la fonction
  - *Passage par référence*: la variable elle-même est envoyée dans la fonction
- Python: la référence à la valeur:
  - *valeur immuable*: la valeur ne peut pas être modifiée par la fonction
  - *valeur variable*: la valeur peut être modifiée par la fonction

```
def myFct(x, y, z):
    x = 56      # new references
    y.append(2) # modify values
    z = [7]     # new references
    print(x)
    print(y)
    print(z)
```

```
a = 77    #immutable
b = [66]  #mutable
c = [1,2] #mutable
myFct(a, b, c)
```

```
56
[66, 2]
[7]
```

```
print(a) #a isn't modified
print(b) #b is modified
print(c) #c isn't modified
```

```
77
[66, 2]
[1, 2]
```

Les fonctions: variables globales

- Les variables extérieures peuvent être utilisées dans une fonction
- Mais elles ne peuvent pas être modifiées
- Pour cela on utilise le mot-clé **global**

```
x = 5
def myFct(y):
    print(x + y)
myFct(10)
```

```
15
```

```
def setx(y):
    x = y
    print('x is now ' + str(y))

setx(10)
print(x)
```

```
x is now 10
5
```

```
def setx(y):
    global x
    x = y
    print('x is now ' + str(y))

setx(10)
print(x)
```

```
x is now 10
10
```

Les fonctions: Bonus

- Les fonctions sont des objets comme les autres:
  - Les assigner à des variables
  - Les passer comme arguments de fonctions

```
def squareFct(x):
    return x ** 2

def doubleFct(x):
    return 2 * x

myFct = squareFct
print(myFct(5))

def computeFct(x, fct=squareFct):
    return fct(x)

print(computeFct(6, doubleFct))
print(computeFct(6, myFct))
```

```
25
12
36
```

- Une fonction peut s'appeler elle-même:

```
def myFct(x):
    if x < 10:
        x = myFct(x+1)
    return x

myFct(0)
```

```
10
```

- *Récursivité*
- **Programmation fonctionnelle**

Gérer les matrices

- En analyse de données, on manipule des matrices
- On peut utiliser une liste de listes:

```
matrix = [[1,2,3], [4,5,6], [7,8,9]]
print('matrix[0][2]: {}'.format(matrix[0][2]))
print("matrix:")
for i in range(3):
    row = ""
    for j in range(3):
        row += '%3i' % matrix[i][j]
    print(row)
```

```
matrix[0][2]: 3
matrix:
  1  2  3
  4  5  6
  7  8  9
```

- Pas évident à utiliser, pas de méthodes ...

Numpy

- Package de calcul numérique de python
- Gère les matrices (et plus généralement les tenseurs)
- **Importer des packages avec python:**

```
import numpy
numpy.array([2,1])
```

```
array([2, 1])
```

```
from numpy import array
array([2,1])
```

```
array([2, 1])
```

```
# Not recommended
from numpy import *
```

```
import numpy as np
np.array([2,1])
```

```
array([2, 1])
```

Numpy: arrays

- Objet de base: `numpy.ndarray`

```
import numpy as np
matrix = (np.arange(9)+1).reshape(3, 3)

print('matrix type:                {}'.format(type(matrix)))
print('matrix number of dimension: {}'.format(matrix.ndim))
print('matrix shape:               {}'.format(matrix.shape))
print('matrix size:                {}'.format(matrix.size))
print('matrix elements type:       {}'.format(matrix.dtype))
print(matrix)
```

```
matrix type:                <class 'numpy.ndarray'>
matrix number of dimension: 2
matrix shape:               (3, 3)
matrix size:                9
matrix elements type:       int64
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Numpy: Créer des *arrays*

- À partir d'une liste python avec `np.array(list)`
- En utilisant `np.zeros`, `np.ones`, `np.empty`, `np.random`
- Avec `np.arange()`, uniquement *1D-arrays*

```
print(np.array([1,2,3]))
np.array([[1,2,3], [4,5,6], [7,8,9]])
np.array([(1,2,3), [4,5,6], (7,8,9)])
```

```
[1 2 3]

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
print(np.arange(3))
print(np.arange(1,10,2))
print(np.arange(0,1,0.3))
```

```
[0 1 2]
[1 3 5 7 9]
[0.  0.3 0.6 0.9]
```

```
print("Generate zeroes filled matrix:")
print(np.zeros((3,2)))
print("\nGenerate ones filled matrix:")
print(np.ones((2,4)))
print("\nGenerate random filled matrix:")
print(np.empty((2,3)))
print("\nGenerate (real) random filled matrix:")
print(np.random.random((2,2)))
```

```
Generate zeroes filled matrix:
[[0. 0.]
 [0. 0.]
 [0. 0.]]

Generate ones filled matrix:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]

Generate random filled matrix:
[[0. 0. 0.]
 [0. 0. 0.]]

Generate (real) random filled matrix:
[[0.37814138 0.60914028]
 [0.46363922 0.48507498]]
```

Numpy: opérations de base

- Les opérations entre *np.array* suivent les lois du calcul matriciel
- Toutes les opérations: +, -, *, /
- Plus des fonctions universelles: `np.exp`, `np.sin`, `np.sqrt`, ...
- **Attention:** * correspond à la multiplication élément par élément
  - Produit matriciel: @ (> python3.5) ou `np.dot()`
- Le résultat d'une opération est un *array* ou un scalaire (int, float, ...)

```
a = np.array([[1,2,3], [2,3,4]])
b = np.arange(1,4) * 100
c = a + b
print(c)

d = a * b
print('{} -> {}'.format(d, type(d)))
```

```
[[101 202 303]
 [102 203 304]]
[[ 100  400  900]
 [ 200  600 1200]] -> <class 'numpy.ndarray'>
```

```
e = a @ b
print(e)

f = np.exp(a)
print(f)
```

```
[1400 2000]
[[ 2.71828183  7.3890561  20.08553692]
 [ 7.3890561  20.08553692 54.59815003]]
```

Numpy: fonctions d'agrégations

- `np.min`, `np.mean`, `np.sum`, ...
- Par défaut: s'applique à toutes les valeurs.
- On peut spécifier selon quel axe les appliquer

```
tot = np.sum(d)
print(tot)

mean = np.mean(f)
std = np.std(f)
print('{} +/- {}'.format(mean, std))
```

```
3400
18.710936317639987 +/- 17.338810526752518
```

```
mean = f.mean()
print('mean = {}'.format(mean))
meanPerRow = f.mean(axis=1)
print('mean per row = {}'.format(meanPerRow))
meanPerCol = f.mean(axis=0)
print('mean per column = {}'.format(meanPerCol))
```

```
mean = 18.710936317639987
mean per row = [10.06429162 27.35758102]
mean per column = [ 5.05366896 13.73729651 37.34184348]
```

Numpy: index et *slicing*

- On accéde aux éléments d'un *1D-array* comme avec une liste python
- Pour les *array* multidimensionnels, le fonctionnement est similaire

```
a = np.random.rand(4,5)
print(a)
print(a[2,4])
```

```
[[0.83365763 0.66020634 0.27622455 0.49378846 0.33601834]
 [0.06984148 0.9841222  0.25837712 0.58801471 0.18511676]
 [0.95084975 0.5684838  0.96051989 0.58512758 0.76710655]
 [0.45737256 0.76542134 0.27405311 0.84011469 0.78727007]]
0.7671065476253957
```

```
print("Get second column:")
print(a[:,1])
print("Get second row:")
print(a[1,:])
print("Get last row:")
print(a[-1])
print("Get first column:")
print(a[...,0])
```

```
Get second column:
[0.66020634 0.9841222  0.5684838  0.76542134]
Get second row:
[0.06984148 0.9841222  0.25837712 0.58801471 0.18511676]
Get last row:
[0.45737256 0.76542134 0.27405311 0.84011469 0.78727007]
Get first column:
[0.83365763 0.06984148 0.95084975 0.45737256]
```

```
for row in a:
    print(row.max())
```

```
0.8336576314111637
0.9841221989206895
0.9605198906974032
0.8401146876391103
```

```
for el in a.flat:
    print(el)
```

```
0.8336576314111637
0.6602063425623879
0.27622455328486917
0.49378846488837935
0.3360183388342397
0.06984147543733843
0.9841221989206895
0.2583771224680298
0.5880147061788084
0.1851167611042518
0.9508497486967813
0.5684838021718104
0.9605198906974032
0.5851275804594694
0.7671065476253957
0.4573725645337252
0.7654213421291775
0.27405310625540724
0.8401146876391103
0.7872700721644015
```

Numpy: manipulation du *shape*

- Il peut être utile de changer le *shape d'un array*
  - Il faut que le nombre d'éléments corresponde
  - `np.reshape()`: retourne un *array* modifié
  - `np.resize()`: modifie l'*array* d'origine
- Transposée d'une matrice: `ndarray.T`

```
a = np.random.random((3,2))
print(a)
print(a.shape)
```

```
[[0.95557862 0.90459495]
 [0.07889465 0.23302825]
 [0.29228728 0.36713287]]
(3, 2)
```

```
print(a.reshape((6,1)))
```

```
[[0.95557862]
 [0.90459495]
 [0.07889465]
 [0.23302825]
 [0.29228728]
 [0.36713287]]
```

- `reshape((x,-1))`, la dimension est automatiquement estimée

```
print(a.reshape((-1,3)))
```

```
[[0.95557862 0.90459495 0.07889465]
 [0.23302825 0.29228728 0.36713287]]
```

```
print(a.T)
print(a.T.shape)
```

```
[[0.95557862 0.07889465 0.29228728]
 [0.90459495 0.23302825 0.36713287]]
(2, 3)
```

```
a.resize((2,3))
print(a)
```

```
[[0.95557862 0.90459495 0.07889465]
 [0.23302825 0.29228728 0.36713287]]
```

Numpy: Concaténer deux *arrays*

- *"coller"* deux matrices **sans ajouter de dimensions**:
  - *"ajouter"* des colonnes: `np.hstack([a, b])`
  - *"ajouter"* des lignes: `np.vstack([a, b])`

```
a = np.random.random((2,2))
b = np.random.random((2,2))
print("Stack verticaly:")
print(np.vstack([a,b]))
print("\nStack horizontaly:")
print(np.hstack([a,b,a]))
```

```
Stack verticaly:
[[0.76375036 0.90950241]
 [0.51441289 0.30522904]
 [0.00237359 0.29705499]
 [0.03062686 0.77044293]]

Stack horizontaly:
[[0.76375036 0.90950241 0.00237359 0.29705499 0.76375036 0.90950241]
 [0.51441289 0.30522904 0.03062686 0.77044293 0.51441289 0.30522904]]
```

Numpy: Concaténer deux *arrays*

- `np.row_stack` = `np.vstack`
- `np.column_stack` = `np.hstack` **2D-array**

```
print(np.column_stack([a, b]))
```

```
[[0.76375036 0.90950241 0.00237359 0.29705499]
 [0.51441289 0.30522904 0.03062686 0.77044293]]
```

```
a = np.array([2, 5])
b = np.array([8, -5])
print('column_stack: 2D')
print(np.column_stack([a, b]))
print('\nhstack: 1D')
print(np.hstack([a, b]))
```

```
column_stack: 2D
[[ 2  8]
 [ 5 -5]]

hstack: 1D
[ 2  5  8 -5]
```

- `np.column_stack` et **1D-array**:

```
print(np.row_stack([a, b]))
```

```
[[ 2  5]
 [ 8 -5]]
```

- Voir aussi: `np.concatenate`:

```
a = np.random.random((2,2))
b = np.random.random((2,2))
print(np.concatenate([a,b], axis=1))
```

```
[[0.00990257 0.83564535 0.77849352 0.50343507]
 [0.82225887 0.46247197 0.23554839 0.94639686]]
```

Numpy: segmenter un *array*

- En utilisant les *slice* (a[3:4,-1,...])
- Ou les méthodes Numpy:
  - `np.hsplit()`, `np.vsplit()`, `np.array_split()`

```
a = np.floor(np.random.random((2,9)) * 100)
print(a)
print('\nSplit matrix:')
print(np.hsplit(a,3))
```

```
[[68. 18. 91. 97. 87. 19. 75. 43. 42.]
 [51. 56.  2. 53. 73. 96. 64.  0. 84.]]

Split matrix:
[array([[68., 18., 91.],
       [51., 56.,  2.]]), array([[97., 87., 19.],
       [53., 73., 96.]]), array([[75., 43., 42.],
       [64.,  0., 84.]])]
```

```
print(np.vsplit(a.T,3))
```

```
[array([[68., 51.],
       [18., 56.],
       [91.,  2.]]), array([[97., 53.],
       [87., 73.],
       [19., 96.]]), array([[75., 64.],
       [43.,  0.],
       [42., 84.]])]
```

```
print(np.array_split(a, 3, axis=1))
```

```
[array([[68., 18., 91.],
       [51., 56.,  2.]]), array([[97., 87., 19.],
       [53., 73., 96.]]), array([[75., 43., 42.],
       [64.,  0., 84.]])]
```

Numpy: copie d'*array*

- `np.array` = objets **variables** python
- On utilise `np.copy()` pour créer une nouvelle référence
- Numpy ajoute la notion de *vue* (*view* ou *shallow copy*):
  - Seules les données de l'objet initial sont modifiées

```python
# No Copy
a = np.array([1, 2, 3])
b = a
print('a = {}, b = {}'.format(a,b))
b[2] = 7
print('a = {}, b = {}'.format(a,b))
b.resize((1,3))
b[0,1] = 0
print('a = {}, b = {}'.format(a,b))
```

```
a = [1 2 3], b = [1 2 3]
a = [1 2 7], b = [1 2 7]
a = [[1 0 7]], b = [[1 0 7]]
```

```python
# Slice
a = np.array([1, 2, 3])
b = a[:2]
print('a = {}, b = {}'.format(a,b))
b[1] = 7
print('a = {}, b = {}'.format(a,b))
```

```
a = [1 2 3], b = [1 2]
a = [1 7 3], b = [1 7]
```

```python
# View
a = np.array([1, 2, 3])
b = a.view()
print('a = {}, b = {}'.format(a,b))
b.resize((1,3))
b[0,1] = 0
print('a = {}, b = {}'.format(a,b))
```

```
a = [1 2 3], b = [1 2 3]
a = [1 0 3], b = [[1 0 3]]
```

```python
# Copy
a = np.array([1, 2, 3])
b = a.copy()
print('a = {}, b = {}'.format(a,b))
b[1] = 0
print('a = {}, b = {}'.format(a,b))
b.resize((1,3))
b[0,1] = 0
print('a = {}, b = {}'.format(a,b))
```

```
a = [1 2 3], b = [1 2 3]
a = [1 2 3], b = [1 0 3]
a = [1 2 3], b = [[1 0 3]]
```

La descente de gradient

Construire un modèle (régression linéaire)

- Notation:
  - $\boldsymbol{m}$ le nombre d'exemples dans notre échantillon
  - $\boldsymbol{n}$ le nombre de variables (*features*)
  - $\boldsymbol{X}$ la matrice des $\boldsymbol{n}$ variables pour les $\boldsymbol{m}$ exemples (x ses éléments)
  - $\boldsymbol{y}$ le vecteur des valeurs à prédire (*vraie* valeurs)
- On cherche à déterminer le modèle pour prédire $\hat{y}$ à partir des x:

$$\hat{y} = h_\theta(x)$$

- On définit le vecteur des paramètres $\boldsymbol{\theta}$ tel que:

$$\hat{y} = \theta_1 x_1 + \cdots + \theta_n x_n = \sum_{i=1}^{n} \theta_i x_i = X \,.\, \theta$$

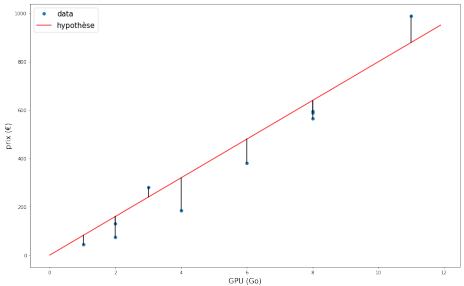
- Rappel math: **fonction linéaire** $f(x) = kx$

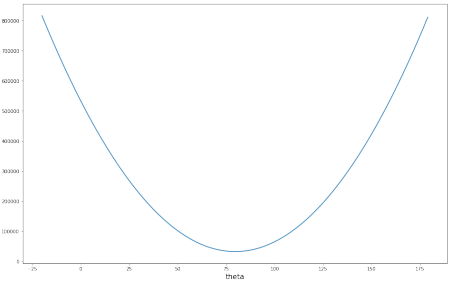
La fonction de coût

- $\mathcal{J}(\theta)$: *véracité* de notre modèle
- Une définition possible: somme quadratique des erreurs

$$\mathcal{J}(\theta) = \frac{1}{2m}\sum_{i=0}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

- On cherche à trouver les valeurs de θ_i qui **minimise** $\mathcal{J}(\theta)$

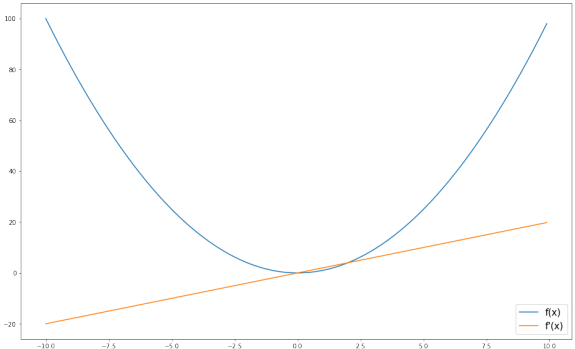




La descente de gradient

- Algorithme pour arriver "*rapidement*" au minimum de $\mathcal{J}(\theta)$
- On va utiliser la *dérivation*: $\frac{d}{d\theta_1}\mathcal{J}(\theta)$:

- Si $\mathcal{J}(\theta)$ est croissant: $\frac{d}{d\theta_1}\mathcal{J}(\theta) > 0$,
- Si $\mathcal{J}(\theta)$ est décroissant: $\frac{d}{d\theta_1}\mathcal{J}(\theta) < 0$



La descente de gradient

- L'algorithme de la descente de gradient s'écrit:

Descente de gradient

Répéter jusqu'à convergence: $\{$

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} \mathcal{J}(\theta)$$

$$\dots$$

$$\theta_n := \theta_n - \alpha \frac{d}{d\theta_n} \mathcal{J}(\theta)$$

$\}$

- α s'appelle le taux d'apprentissage (*learning rate*) et c'est le **seul** paramètre de l'algorithme.
- On va itérativement modifier la valeur de θ_1 en fonction de la dérivée de $\mathcal{J}(\theta)$, jusqu'à minimiser $\mathcal{J}(\theta)$ (*convergence*).

La descente de gradient

- Dérivons donc notre fonction de coût:

$$\mathcal{J}(\theta) = \frac{1}{2m}\sum_{i=0}^{m}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m}\sum_{i=0}^{m}(\sum_{j=0}^{n}\theta_j x_j^{(i)} - y^{(i)})^2$$

$$\frac{d}{d\theta_j}\mathcal{J}(\theta) = \frac{1}{m}\sum_{i=0}^{m}(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$$

- Un peu de *hand-tunning*:
  - La valeur idéale du learning rate (α) doit être testée (en général: ~ 0.03)
  - La précision nous servira à arrêter la descente de gradient: $\epsilon = 0.0001$