Apprendre X en Y minutes (/)

Où X=Python

Récupérer le code : learnpython-fr.py (/docs/files/learnpython-fr.py)

Python a été créé par Guido Van Rossum au début des années 90. C'est maintenant un des langages les plus populaires. Je suis tombé amoureux de Python pour la clarté de sa syntaxe. C'est tout simplement du pseudo-code exécutable.

L'auteur original apprécierait les retours (en anglais): vous pouvez le contacter sur Twitter à @louiedinh (http://twitter.com/louiedinh) ou par mail à l'adresse louiedinh [at] [google's email service]

Note : Cet article s'applique spécifiquement à Python 3. Jettez un coup d'oeil ici (http://learnxinyminutes.com/docs/fr-fr/python-fr/) pour apprendre le vieux Python 2.7

```
# Un commentaire d'une ligne commence par un dièse

""" Les chaînes de caractères peuvent être écrites
    avec 3 guillemets doubles ("), et sont souvent
    utilisées comme des commentaires.
"""

####################################################
## 1. Types de données primaires et opérateurs
####################################################

# On a des nombres
3  # => 3

# Les calculs sont ce à quoi on s'attend
1 + 1  # => 2
8 - 1  # => 7
10 * 2  # => 20

# Sauf pour la division qui retourne un float (nombre à virgule
flottante)
35 / 5  # => 7.0

# Résultats de divisions entières tronqués pour les nombres positifs et
négatifs
```

```
5 // 3     # => 1
5.0 // 3.0 # => 1.0 # works on floats too
-5 // 3  # => -2
-5.0 // 3.0 # => -2.0

# Quand on utilise un float, le résultat est un float
3 * 2.0 # => 6.0

# Modulo (reste de la division)
7 % 3 # => 1

# Exponentiation (x**y, x élevé à la puissance y)
2**4 # => 16

# Forcer la priorité de calcul avec des parenthèses
(1 + 3) * 2  # => 8

# Les valeurs booléennes sont primitives
True
False

# Négation avec not
not True  # => False
not False  # => True

# Opérateurs booléens
# On note que "and" et "or" sont sensibles à la casse
True and False #=> False
False or True #=> True

# Utilisation des opérations booléennes avec des entiers :
0 and 2 #=> 0
-5 or 0 #=> -5
0 == False #=> True
2 == True #=> False
1 == True #=> True

# On vérifie une égalité avec ==
1 == 1  # => True
2 == 1  # => False

# On vérifie une inégalité avec !=
1 != 1  # => False
2 != 1  # => True

# Autres opérateurs de comparaison
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# On peut enchaîner les comparaisons
1 < 2 < 3  # => True
```

```
2 < 3 < 2  # => False

# (is vs. ==) is vérifie si deux variables pointent sur le même objet,
mais == vérifie
# si les objets ont la même valeur.
a = [1, 2, 3, 4] # a pointe sur une nouvelle liste, [1, 2, 3, 4]
b = a # b pointe sur a
b is a # => True, a et b pointent sur le même objet
b == a # => True, les objets a et b sont égaux
b = [1, 2, 3, 4] # b pointe sur une nouvelle liste, [1, 2, 3, 4]
b is a # => False, a et b ne pointent pas sur le même objet
b == a # => True, les objets a et b ne pointent pas sur le même objet

# Les chaînes (ou strings) sont créées avec " ou '
"Ceci est une chaine"
'Ceci est une chaine aussi.'

# On peut additionner des chaînes aussi ! Mais essayez d'éviter de le
faire.
"Hello " + "world!"  # => "Hello world!"
# On peut aussi le faire sans utiliser '+'
"Hello " "world!"  # => "Hello world!"

# On peut traîter une chaîne comme une liste de caractères
"This is a string"[0]  # => 'T'

# .format peut être utilisé pour formatter des chaînes, comme ceci:
"{} peuvent etre {}".format("Les chaînes", "interpolées")

# On peut aussi réutiliser le même argument pour gagner du temps.
"{0} be nimble, {0} be quick, {0} jump over the {1}".format("Jack",
"candle stick")
#=> "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# On peut aussi utiliser des mots clés pour éviter de devoir compter.
"{name} wants to eat {food}".format(name="Bob", food="lasagna") #=> "Bob
wants to eat lasagna"

# Il est également possible d'utiliser les f-strings depuis Python 3.6
(https://docs.python.org/3/whatsnew/3.6.html#pep-498-formatted-string-
literals)
name = "Fred"
f"Il a dit que son nom est {name}." #=> "Il a dit que son nom est Fred."

# Si votre code doit aussi être compatible avec Python 2.5 et moins,
# vous pouvez encore utiliser l'ancienne syntaxe :
"Les %s peuvent être %s avec la %s méthode" % ("chaînes", "interpolées",
"vieille")


# None est un objet
None  # => None
```

```
# N'utilisez pas "==" pour comparer des objets à None
# Utilisez plutôt "is". Cela permet de vérifier l'égalité de l'identité
des objets.
"etc" is None  # => False
None is None  # => True

# None, 0, and les strings/lists/dicts (chaînes/listes/dictionnaires)
valent False lorsqu'ils sont convertis en booléens.
# Toutes les autres valeurs valent True
bool(0)  # => False
bool("")  # => False
bool([]) #=> False
bool({}) #=> False


####################################################
## 2. Variables et Collections
####################################################

# Python a une fonction print pour afficher du texte
print("I'm Python. Nice to meet you!")

# Par défaut, la fonction print affiche aussi une nouvelle ligne à la
fin.
# Utilisez l'argument optionnel end pour changer ce caractère de fin.
print("Hello, World", end="!") # => Hello, World!

# Pas besoin de déclarer des variables avant de les définir.
# La convention est de nommer ses variables avec des
minuscules_et_underscores
some_var = 5
some_var  # => 5

# Tenter d'accéder à une variable non définie lève une exception.
# Voir Structures de contrôle pour en apprendre plus sur le traitement
des exceptions.
une_variable_inconnue  # Lève une NameError

# Les listes permettent de stocker des séquences
li = []
# On peut initialiser une liste pré-remplie
other_li = [4, 5, 6]

# On ajoute des objets à la fin d'une liste avec .append
li.append(1)    # li vaut maintenant [1]
li.append(2)    # li vaut maintenant [1, 2]
li.append(4)    # li vaut maintenant [1, 2, 4]
li.append(3)    # li vaut maintenant [1, 2, 4, 3]
# On enlève le dernier élément avec .pop
li.pop()        # => 3 et li vaut maintenant [1, 2, 4]
# Et on le remet
li.append(3)    # li vaut de nouveau [1, 2, 4, 3]
```

```
# Accès à un élément d'une liste :
li[0]  # => 1
# Accès au dernier élément :
li[-1]  # => 3

# Accéder à un élément en dehors des limites lève une IndexError
li[4]  # Lève une IndexError

# On peut accéder à une intervalle avec la syntaxe "slice"
# (c'est un rang du type "fermé/ouvert")
li[1:3]  # => [2, 4]
# Omettre les deux premiers éléments
li[2:]  # => [4, 3]
# Prendre les trois premiers
li[:3]  # => [1, 2, 4]
# Sélectionner un élément sur deux
li[::2]   # =>[1, 4]
# Avoir une copie de la liste à l'envers
li[::-1]   # => [3, 4, 2, 1]
# Pour des "slices" plus élaborées :
# li[debut:fin:pas]

# Faire une copie d'une profondeur de un avec les "slices"
li2 = li[:] # => li2 = [1, 2, 4, 3] mais (li2 is li) vaut False.

# Enlever des éléments arbitrairement d'une liste
del li[2]   # li is now [1, 2, 3]

# On peut additionner des listes
# Note: les valeurs de li et other_li ne sont pas modifiées.
li + other_li   # => [1, 2, 3, 4, 5, 6]

# Concaténer des listes avec "extend()"
li.extend(other_li)   # Maintenant li contient [1, 2, 3, 4, 5, 6]

# Vérifier la présence d'un objet dans une liste avec "in"
1 in li   # => True

# Examiner la longueur avec "len()"
len(li)   # => 6


# Les tuples sont comme des listes mais sont immuables.
tup = (1, 2, 3)
tup[0]   # => 1
tup[0] = 3  # Lève une TypeError

# Note : un tuple de taille un doit avoir une virgule après le dernier
élément,
# mais ce n'est pas le cas des tuples d'autres tailles, même zéro.
type((1))  # => <class 'int'>
type((1,)) # => <class 'tuple'>
type(())   # => <class 'tuple'>
```

```
# On peut utiliser la plupart des opérations des listes sur des tuples.
len(tup)   # => 3
tup + (4, 5, 6)   # => (1, 2, 3, 4, 5, 6)
tup[:2]   # => (1, 2)
2 in tup   # => True

# Vous pouvez décomposer des tuples (ou des listes) dans des variables
a, b, c = (1, 2, 3)     # a vaut 1, b vaut 2 et c vaut 3
# Les tuples sont créés par défaut sans parenthèses
d, e, f = 4, 5, 6
# Voyez comme il est facile d'intervertir deux valeurs :
e, d = d, e     # d vaut maintenant 5 et e vaut maintenant 4


# Créer un dictionnaire :
empty_dict = {}
# Un dictionnaire pré-rempli :
filled_dict = {"one": 1, "two": 2, "three": 3}

# Note : les clés des dictionnaires doivent être de types immuables.
# Elles doivent être convertibles en une valeur constante pour une
recherche rapide.
# Les types immuables incluent les ints, floats, strings et tuples.
invalid_dict = {[1,2,3]: "123"} # => Lève une TypeError: unhashable type:
'list'
valid_dict = {(1,2,3):[1,2,3]}  # Par contre, les valeurs peuvent être de
tout type.

# On trouve une valeur avec []
filled_dict["one"]   # => 1

# On obtient toutes les clés sous forme d'un itérable avec "keys()" Il
faut l'entourer
# de list() pour avoir une liste Note: l'ordre n'est pas garanti.
list(filled_dict.keys())   # => ["three", "two", "one"]


# On obtient toutes les valeurs sous forme d'un itérable avec "values()".
# Là aussi, il faut utiliser list() pour avoir une liste.
# Note : l'ordre n'est toujours pas garanti.
list(filled_dict.values())   # => [3, 2, 1]


# On vérifie la présence d'une clé dans un dictionnaire avec "in"
"one" in filled_dict   # => True
1 in filled_dict   # => False

# L'accès à une clé non-existente lève une KeyError
filled_dict["four"]   # KeyError

# On utilise "get()" pour éviter la KeyError
filled_dict.get("one")   # => 1
```

```
filled_dict.get("four")   # => None
# La méthode get accepte une valeur de retour par défaut en cas de valeur
non-existante.
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)   # => 4

# "setdefault()" insère une valeur dans un dictionnaire si la clé n'est
pas présente.
filled_dict.setdefault("five", 5)  # filled_dict["five"] devient 5
filled_dict.setdefault("five", 6)  # filled_dict["five"] est toujours 5

# Ajouter à un dictionnaire
filled_dict.update({"four":4}) #=> {"one": 1, "two": 2, "three": 3,
"four": 4}
#filled_dict["four"] = 4  # une autre méthode

# Enlever des clés d'un dictionnaire avec del
del filled_dict["one"]  # Enlever la clé "one" de filled_dict.


# Les sets stockent des ensembles
empty_set = set()
# Initialiser un set avec des valeurs. Oui, ça ressemble aux
dictionnaires, désolé.
some_set = {1, 1, 2, 2, 3, 4}   # some_set est maintenant {1, 2, 3, 4}

# Comme les clés d'un dictionnaire, les éléments d'un set doivent être
immuables.
invalid_set = {[1], 1} # => Lève une TypeError: unhashable type: 'list'
valid_set = {(1,), 1}

# On peut changer un set :
filled_set = some_set

# Ajouter un objet au set :
filled_set.add(5)   # filled_set vaut maintenant {1, 2, 3, 4, 5}

# Chercher les intersections de deux sets avec &
other_set = {3, 4, 5, 6}
filled_set & other_set   # => {3, 4, 5}

# On fait l'union de sets avec |
filled_set | other_set   # => {1, 2, 3, 4, 5, 6}

# On fait la différence de deux sets avec -
{1, 2, 3, 4} - {2, 3, 5}   # => {1, 4}

# On vérifie la présence d'un objet dans un set avec in
2 in filled_set   # => True
10 in filled_set   # => False
```

```
####################################################
## 3. Structures de contrôle et Itérables
####################################################

# On crée juste une variable
some_var = 5

# Voici une condition "si". L'indentation est significative en Python!
# Affiche: "some_var is smaller than 10"
if some_var > 10:
    print("some_var is totally bigger than 10.")
elif some_var < 10:    # La clause elif ("sinon si") est optionelle
    print("some_var is smaller than 10.")
else:                  # La clause else ("sinon") l'est aussi.
    print("some_var is indeed 10.")


"""
Les boucles "for" itèrent sur une liste
Affiche:
    chien est un mammifère
    chat est un mammifère
    souris est un mammifère
"""
for animal in ["chien", "chat", "souris"]:
    # On peut utiliser format() pour interpoler des chaînes formattées
    print("{} est un mammifère".format(animal))

"""
"range(nombre)" retourne un itérable de nombres
de zéro au nombre donné
Affiche:
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
"range(debut, fin)" retourne un itérable de nombre
de debut à fin.
Affiche:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print(i)

"""
```

```
"range(debut, fin, pas)" retourne un itérable de nombres
de début à fin en incrémentant de pas.
Si le pas n'est pas indiqué, la valeur par défaut est 1.
Affiche:
    4
    6
    8
"""
for i in range(4, 8, 2):
    print(i)
"""

Les boucles "while" bouclent jusqu'à ce que la condition devienne fausse.
Affiche:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print(x)
    x += 1  # Raccourci pour x = x + 1

# On gère les exceptions avec un bloc try/except
try:
    # On utilise "raise" pour lever une erreur
    raise IndexError("Ceci est une erreur d'index")
except IndexError as e:
    pass    # Pass signifie simplement "ne rien faire". Généralement, on
gère l'erreur ici.
except (TypeError, NameError):
    pass    # Si besoin, on peut aussi gérer plusieurs erreurs en même
temps.
else:   # Clause optionelle des blocs try/except. Doit être après tous
les except.
    print("Tout va bien!")   # Uniquement si aucune exception n'est
levée.
finally: #  Éxécuté dans toutes les circonstances.
    print("On nettoie les ressources ici")

# Au lieu de try/finally pour nettoyer les ressources, on peut utiliser
with
with open("myfile.txt") as f:
    for line in f:
        print(line)

# Python offre une abstraction fondamentale : l'Iterable.
# Un itérable est un objet pouvant être traîté comme une séquence.
# L'objet retourné par la fonction range() est un itérable.

filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
```

```
print(our_iterable) #=> range(1,10). C'est un objet qui implémente
l'interface Iterable

# On peut boucler dessus
for i in our_iterable:
    print(i)    # Affiche one, two, three

# Cependant, on ne peut pas accéder aux éléments par leur adresse.
our_iterable[1]  # Lève une TypeError

# Un itérable est un objet qui sait créer un itérateur.
our_iterator = iter(our_iterable)

# Notre itérateur est un objet qui se rappelle de notre position quand on
le traverse.
# On passe à l'élément suivant avec "next()".
next(our_iterator)  #=> "one"

# Il garde son état quand on itère.
next(our_iterator)  #=> "two"
next(our_iterator)  #=> "three"

# Après que l'itérateur a retourné toutes ses données, il lève une
exception StopIterator
next(our_iterator) # Lève une StopIteration

# On peut mettre tous les éléments d'un itérateur dans une liste avec
list()
list(filled_dict.keys())  #=> Returns ["one", "two", "three"]


####################################################
## 4. Fonctions
####################################################

# On utilise "def" pour créer des fonctions
def add(x, y):
    print("x est {} et y est {}".format(x, y))
    return x + y    # On retourne une valeur avec return

# Appel d'une fonction avec des paramètres :
add(5, 6)   # => affiche "x est 5 et y est 6" et retourne 11

# Une autre manière d'appeler une fonction : avec des arguments
add(y=6, x=5)   # Les arguments peuvent être dans n'importe quel ordre.

# Définir une fonction qui prend un nombre variable d'arguments
def varargs(*args):
    return args

varargs(1, 2, 3)   # => (1, 2, 3)

# On peut aussi définir une fonction qui prend un nombre variable de
```

```
paramètres.
def keyword_args(**kwargs):
    return kwargs

# Appelons la pour voir ce qu'il se passe :
keyword_args(big="foot", loch="ness")   # => {"big": "foot", "loch":
"ness"}


# On peut aussi faire les deux à la fois :
def all_the_args(*args, **kwargs):
    print(args)
    print(kwargs)
"""
all_the_args(1, 2, a=3, b=4) affiche:
    (1, 2)
    {"a": 3, "b": 4}
"""

# En appelant des fonctions, on peut aussi faire l'inverse :
# utiliser * pour étendre un tuple de paramètres
# et ** pour étendre un dictionnaire d'arguments.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)   # équivalent à foo(1, 2, 3, 4)
all_the_args(**kwargs)   # équivalent à foo(a=3, b=4)
all_the_args(*args, **kwargs)   # équivalent à foo(1, 2, 3, 4, a=3, b=4)

# Retourne plusieurs valeurs (avec un tuple)
def swap(x, y):
    return y, x # Retourne plusieurs valeurs avec un tuple sans
parenthèses.
                # (Note: on peut aussi utiliser des parenthèses)

x = 1
y = 2
x, y = swap(x, y) # => x = 2, y = 1
# (x, y) = swap(x,y) # Là aussi, rien ne nous empêche d'ajouter des
parenthèses

# Portée des fonctions :
x = 5

def setX(num):
    # La variable locale x n'est pas la même que la variable globale x
    x = num # => 43
    print (x) # => 43

def setGlobalX(num):
    global x
    print (x) # => 5
    x = num # la variable globale x est maintenant 6
    print (x) # => 6
```

```
setX(43)
setGlobalX(6)


# Python a des fonctions de première classe
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13

# Mais aussi des fonctions anonymes
(lambda x: x > 2)(3)   # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1) # => 5

# TODO - Fix for iterables
# Il y a aussi des fonctions de base
map(add_10, [1, 2, 3])   # => [11, 12, 13]
map(max, [1, 2, 3], [4, 2, 1])   # => [4, 2, 3]

filter(lambda x: x > 5, [3, 4, 5, 6, 7])   # => [6, 7]

# On peut utiliser les compréhensions de listes pour de jolies maps et
filtres.
# Une compréhension de liste stocke la sortie comme une liste qui peut
elle même être une liste imbriquée.
[add_10(i) for i in [1, 2, 3]]  # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]   # => [6, 7]

####################################################
## 5. Classes
####################################################


# On utilise l'opérateur "class" pour définir une classe
class Human:

    # Un attribut de la classe. Il est partagé par toutes les instances
de la classe.
    species = "H. sapiens"

    # L'initialiseur de base. Il est appelé quand la classe est
instanciée.
    # Note : les doubles underscores au début et à la fin sont utilisés
pour
    # les fonctions et attributs utilisés par Python mais contrôlés par
l'utilisateur.
    # Les méthodes (ou objets ou attributs) comme: __init__, __str__,
    # __repr__ etc. sont appelés méthodes magiques.
    # Vous ne devriez pas inventer de noms de ce style.
```

```
    def __init__(self, name):
        # Assigner l'argument à l'attribut de l'instance
        self.name = name

    # Une méthode de l'instance. Toutes prennent "self" comme premier
argument.
    def say(self, msg):
        return "{name}: {message}".format(name=self.name, message=msg)

    # Une méthode de classe est partagée avec entre les instances
    # Ils sont appelés avec la classe comme premier argument
    @classmethod
    def get_species(cls):
        return cls.species

    # Une méthode statique est appelée sans référence à une instance ni à
une classe.
    @staticmethod
    def grunt():
        return "*grunt*"


# Instantier une classe
i = Human(name="Ian")
print(i.say("hi"))     # affiche "Ian: hi"

j = Human("Joel")
print(j.say("hello"))  # affiche "Joel: hello"

# Appeller notre méthode de classe
i.get_species()   # => "H. sapiens"

# Changer les attributs partagés
Human.species = "H. neanderthalensis"
i.get_species()   # => "H. neanderthalensis"
j.get_species()   # => "H. neanderthalensis"

# Appeller la méthode statique
Human.grunt()   # => "*grunt*"


####################################################
## 6. Modules
####################################################

# On peut importer des modules
import math
print(math.sqrt(16))  # => 4.0

# On peut importer des fonctions spécifiques d'un module
from math import ceil, floor
print(ceil(3.7))  # => 4.0
print(floor(3.7))   # => 3.0
```

```
# On peut importer toutes les fonctions d'un module
# Attention: ce n'est pas recommandé.
from math import *

# On peut raccourcir un nom de module
import math as m
math.sqrt(16) == m.sqrt(16)   # => True

# Les modules Python sont juste des fichiers Python.
# Vous pouvez écrire les vôtres et les importer. Le nom du module
# est le nom du fichier.

# On peut voir quels fonctions et objets un module définit
import math
dir(math)


####################################################
## 7. Avancé
####################################################

# Les générateurs aident à faire du code paresseux (lazy)
def double_numbers(iterable):
    for i in iterable:
        yield i + i

# Un générateur crée des valeurs à la volée.
# Au lieu de générer et retourner toutes les valeurs en une fois, il en
crée une à chaque
# itération.  Cela signifie que les valeurs supérieures à 30 ne seront
pas traîtées par
# double_numbers.
# Note : range est un générateur aussi.
# Créer une liste 1-900000000 prendrait beaucoup de temps
# On met un underscore à la fin d'un nom de variable normalement réservé
par Python.
range_ = range(1, 900000000)
# Double tous les nombres jusqu'à ce qu'un nombre >=30 soit trouvé
for i in double_numbers(range_):
    print(i)
    if i >= 30:
        break


# Decorateurs
# Dans cet exemple, beg enveloppe say
# Beg appellera say. Si say_please vaut True le message retourné sera
changé
from functools import wraps


def beg(target_function):
```

```
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper


@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please


print(say())  # affiche Can you buy me a beer?
print(say(say_please=True))  # affiche Can you buy me a beer? Please! I
am poor :(
```

Prêt pour encore plus ?

En ligne et gratuit (en anglais)

- Automate the Boring Stuff with Python (https://automatetheboringstuff.com)
- Learn Python The Hard Way (http://learnpythonthehardway.org/book/)
- Dive Into Python (http://www.diveintopython.net/)
- Ideas for Python Projects (http://pythonpracticeprojects.com)
- The Official Docs (http://docs.python.org/3/)
- Hitchhiker's Guide to Python (http://docs.python-guide.org/en/latest/)
- A Crash Course in Python for Scientists (http://nbviewer.ipython.org/5920182)
- Python Course (http://www.python-course.eu/index.php)
- First Steps With Python (https://realpython.com/learn/python-first-steps/)

En ligne et gratuit (en français)

- Le petit guide des batteries à découvrir (https://he-arc.github.io/livre-python/)

Livres (en anglais)

- Programming Python (http://www.amazon.com/gp/product/0596158106/ref=as_li_qf_sp_asin_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0596158106&linkCode=as2&tag=homebits04-20)
- Dive Into Python (http://www.amazon.com/gp/product/1441413022/ref=as_li_tf_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=1441413022&linkCode=as2&tag=homebits04-

20)
- Python Essential Reference (http://www.amazon.com/gp/product/0672329786/ref=as_li_tf_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0672329786&linkCode=as2&tag=homebits04-20)

Vous avez une suggestion ? Peut-être une correction ? Ouvrez un ticket (https://github.com/adambard/learnxinyminutes-docs/issues/new) sur Github, ou faites vous-même une pull request (https://github.com/adambard/learnxinyminutes-docs/edit/master/fr-fr/python-fr.html.markdown) !

Version originale par Louie Dinh, mis à jour par 2 contributeur(s) (https://github.com/adambard/learnxinyminutes-docs/blame/master/fr-fr/python-fr.html.markdown).

Translated by: Gnomino (https://github.com/Gnomino) Julien M'Poy (http://github.com/groovytron)