

# Python à toute vitesse

João Ventura

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
	Pour installer sur Windows . . . . .	2
	Pour installer sur MacOS . . . . .	4
	Pour installer sur Linux . . . . .	5
<b>2</b>	<b>Nombres et chaînes de caractères</b>	<b>6</b>
	Exercices avec les nombres . . . . .	7
	Exercices avec des chaînes de caractères . . . . .	7
<b>3</b>	<b>Les listes</b>	<b>9</b>
	Exercices avec des listes . . . . .	10
	Les compréhensions de liste . . . . .	10
	Exercices avec des compréhensions de liste . . . . .	10
	Exercices avec le module math . . . . .	12
	Exercices avec les fonctions . . . . .	12
	Les fonctions récursives . . . . .	12
	Exercices avec les fonctions récursives . . . . .	13
	Exercices avec la boucle for . . . . .	15
	Exercices avec l'expression while . . . . .	16
<b>4</b>	<b>Les Dictionnaires</b>	<b>17</b>
	Exercices avec les dictionnaires . . . . .	18
	Exercices avec des sous-dictionnaires . . . . .	19
	Exercices avec des classes . . . . .	20
	L'Héritage de classe . . . . .	20
	Exercices avec l'héritage . . . . .	21
	Les Classes Itérateurs . . . . .	22
	Exercices avec les itérateurs . . . . .	23
	Exercices avec les générateurs . . . . .	24
	Exercices avec les coroutines . . . . .	25
	Les Pipelines . . . . .	25
	Exercices avec des pipelines de coroutines . . . . .	28
	Exercices avec asyncio . . . . .	31

# Chapter 1

## Introduction

Ce livre vise à enseigner le langage de programmation Python en suivant une approche pratique. Sa méthode est simple: après une courte introduction à chaque sujet, le lecteur est invité à en apprendre davantage en résolvant les exercices proposés.

Ces exercices ont été abondamment utilisés dans mes cours de développement web et de programmation distribuée à l'Ecole Supérieure de Technologie de Setúbal. Grâce à ces exercices, les élèves sont compétents en Python en moins d'un mois. En fait, les étudiants du cours de programmation distribuée, cours enseigné dans la seconde année du diplôme d'ingénieur informaticien, sont familiers avec la syntaxe après deux semaines, et sont capables d'implémenter une application client-serveur distribuée avec sockets la troisième semaine.

Ce livre est un travail en cours d'élaboration et il est possible qu'il contienne quelques fautes d'orthographe qui seront corrigées dans le futur. Néanmoins il est rendu disponible dès maintenant de manière à pouvoir être utile à quiconque souhaite l'utiliser. J'espère sincèrement que vous en tirerez quelque chose de positif.

Le code source de ce livre est disponible sur github (<https://github.com/joaovventura/full-speed-python>). J'accepterai avec plaisir toute pull request visant à corriger l'orthographe ou la grammaire, à suggérer de nouveaux exercices ou à clarifier le contenu actuel.

Bien à vous,

João Ventura - Professeur Adjoint à l'Ecole supérieure de Technologie de Setúbal. #  
Installation

Dans ce chapitre nous installerons et lancerons l'interpréteur Python sur votre ordinateur.

## Pour installer sur Windows

1. Téléchargez la dernière version de Python 3 pour Windows à <https://www.python.org/downloads/windows/> et lancez l'installateur. Au moment d'écrire ce livre, la dernière version est Python 3.7.4.

2. Assurez-vous de sélectionner les paramètres “Install launcher for all users” et “Add Python to PATH”, et choisissez “Customize installation”.



Figure 1.1: Windows installation

3. Sur l’écran suivant, “Optional Features”, vous pouvez tout installer, mais il est surtout essentiel d’installer “pip” et “pylauncher (for all users)”. Pip est le gestionnaire de paquets de Python qui vous permet d’installer de nombreux paquets et bibliothèques.
4. Dans les Options Avancées, assurez-vous de sélectionner “Add Python to environment variables”. Aussi, je vous suggère de changer l’emplacement de l’installation vers quelque chose comme C:\Python36\. Ce sera plus facile pour vous de retrouver l’installation de Python en cas de problème.

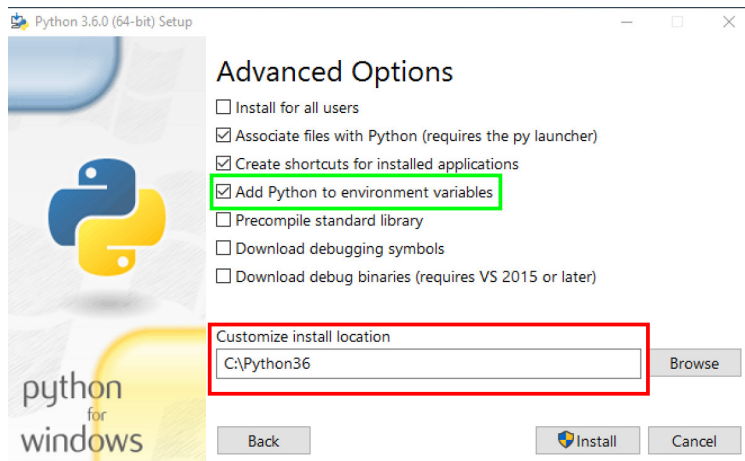


Figure 1.2: Windows installation

5. Finalement, permettez à Python d’utiliser plus de 260 caractères dans le système de fichiers en sélectionnant “Disable path length limit” et fermez la boîte de dialogue d’installation
6. Maintenant, ouvrez la ligne de commande (cmd) et exécutez “python” ou “python3”. Si tout est correctement installé, vous devriez voir la REPL de Python. La REPL (pour Read, Evaluate, Print Loop) est un environnement que vous pouvez utiliser

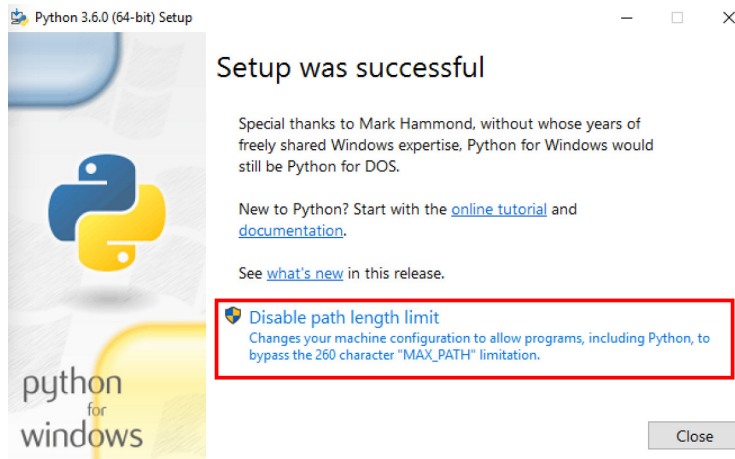


Figure 1.3: Windows installation

pour programmer des petits bouts de code en Python. Tapez `exit()` pour quitter la REPL.

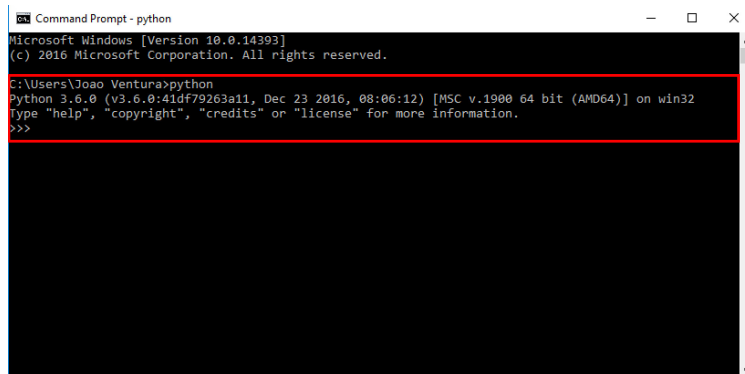
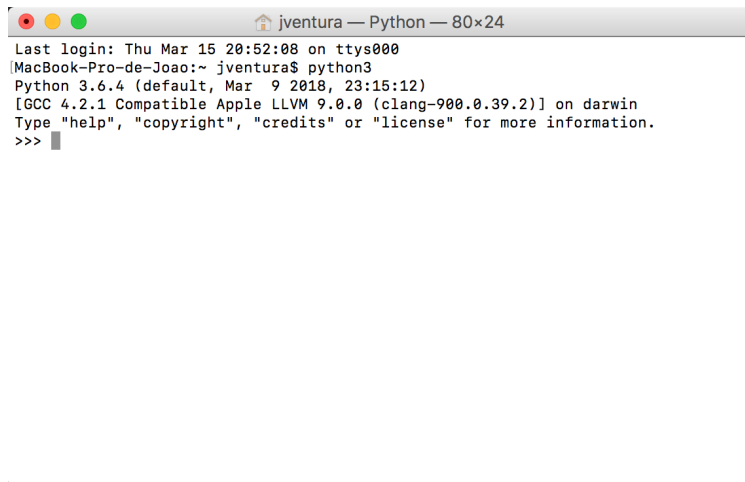


Figure 1.4: Python REPL

## Pour installer sur MacOS

Vous pouvez télécharger les derniers exécutables depuis <https://www.python.org/downloads/mac-osx/>. Assurez-vous de télécharger les dernières versions pour Python 3 (Python 3.7.4 au moment d’écrire ces lignes). Vous pouvez également utiliser Homebrew, un gestionnaire de paquets pour MacOS (<https://brew.sh/>). Pour installer la dernière version de Python 3 avec Homebrew, tapez “`brew install python3`” sur votre terminal. Une autre option est d’utiliser le gestionnaire de paquets MacPorts (<https://www.macports.org/>) et la commande “`port install python36`”.

Finalement, ouvrez le terminal, exécutez `python3` et vous devriez voir la REPL de Python comme expliqué plus haut. Pressez `Ctrl+D` ou tapez `exit()` pour quitter la REPL.

A terminal window titled 'jventura — Python — 80x24'. The output shows the last login time, the command 'python3' being executed, and the resulting Python version (3.6.4) and compiler information (GCC 4.2.1, LLVM 9.0.0) on a Darwin system. The prompt is '>>>' with a cursor.

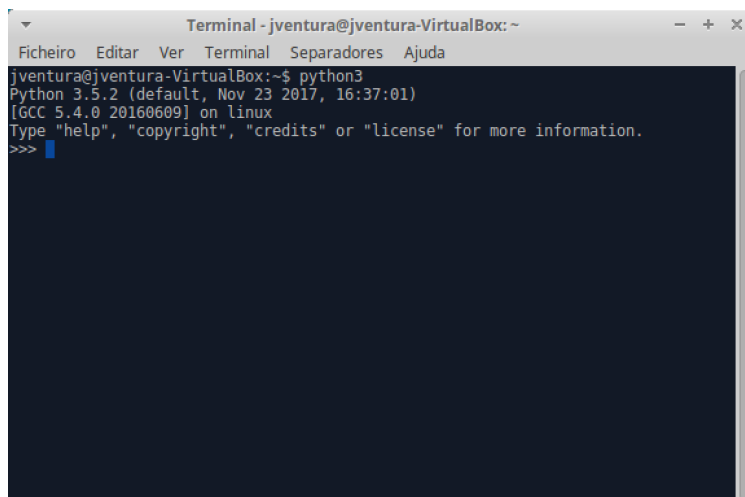
```
jventura — Python — 80x24
Last login: Thu Mar 15 20:52:08 on ttys000
MacBook-Pro-de-Joao:~ jventura$ python3
Python 3.6.4 (default, Mar  9 2018, 23:15:12)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figure 1.5: Python REPL

## Pour installer sur Linux

Pour installer Python sur Linux, vous pouvez télécharger les derniers fichiers sources de Python 3 depuis <https://www.python.org/downloads/source/> ou utiliser le gestionnaire de paquets (apt-get, aptitude, synaptic ou autres) pour l'installer. Pour vous assurer d'avoir Python 3 installé sur votre système, tapez `python3 --version` dans votre terminal.

Finalement, ouvrez le terminal, exécutez `python3` et vous devriez voir la REPL de Python comme dans l'image suivante. Pressez Ctrl+D ou tapez `exit()` pour quitter la REPL.

A terminal window titled 'Terminal - jventura@jventura-VirtualBox: ~'. The output shows the command 'python3' being executed, the resulting Python version (3.5.2) and compiler information (GCC 5.4.0) on a Linux system. The prompt is '>>>' with a cursor.

```
Terminal - jventura@jventura-VirtualBox: ~
Ficheiro  Editar  Ver  Terminal  Separadores  Ajuda
jventura@jventura-VirtualBox:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figure 1.6: Python REPL

# Chapter 2

## Nombres et chaînes de caractères

Dans ce chapitre nous travaillerons avec les types de données les plus basiques, les nombres et les chaînes de caractères. Démarrez votre REPL Python et écrivez ce qui suit:

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> b = 2.5
>>> type(b)
<class 'float'>
```

Pour l'essentiel, vous déclarez deux variables (nommées "a" et "b") qui vont contenir des nombres: la variable "a" est un nombre entier tandis que la variable "b" est un nombre réel. Nous pouvons maintenant utiliser nos variables ou n'importe quels autres nombres pour effectuer des calculs:

```
>>> a + b
4.5
>>> (a + b) * 2
9.0
>>> 2 + 2 + 4 - 2/3
7.333333333333333
```

Python gère également les chaînes de caractères. Les chaînes de caractères sont des suites de lettres (comme les mots) et peuvent être définies en utilisant des guillemets simples ou doubles:

```
>>> hi = "hello"
>>> hi
'hello'
>>> bye = 'goodbye'
>>> bye
'goodbye'
```

Vous pouvez additionner les chaînes de caractères pour les concaténer, mais vous ne pouvez pas mélanger différents types de données, tels que les chaînes de caractères et les nombres.

```
>>> hi + "world"
'helloworld'
>>> "Hello" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Néanmoins, la multiplication fonctionne comme répétition:

```
>>> "Hello" * 3
'HelloHelloHello'
```

## Exercices avec les nombres

1. Essayez les calculs mathématiques suivants et devinez ce qui se passe:  $((3 / 2))$ ,  $((3 // 2))$ ,  $((3 \% 2))$ ,  $((3**2))$ .

Suggestion: Ouvrez la documentation de référence de la bibliothèque Python à <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>.

2. Calculez la moyenne des séquences de nombres suivantes: (2, 4), (4, 8, 9), (12, 14/6, 15)
3. Le volume d'une sphère est donné par  $(4/3 * \pi * r^3)$ . Calculez le volume d'une sphère de radius 5. Suggestion: créez une variable nommée "pi" avec une valeur de 3.145.
4. Utilisez l'opérateur modulo (%) pour voir lesquels the nombres suivants sont pairs: (1, 5, 20, 60/7).

Suggestion: le reste de la division  $(x/2)$  est toujours zéro lorsque (x) est pair.

5. Trouvez des valeurs pour (x) et (y) telles que  $(x < 1/3 < y)$  retourne "True" sur la REPL Python. Suggestion: essayez  $(0 < 1/3 < 1)$  sur la REPL.

## Exercices avec des chaînes de caractères

En utilisant la documentation de Python sur les chaînes de caractères (<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>), effectuez les exercices suivants:

1. Initialisez la chaîne "abc" dans une variable nommée "s":
  1. Utilisez une fonction pour obtenir la longueur de la chaîne.
  2. Ecrivez la séquence d'opérations nécessaire pour transformer la chaîne "abc" en "aaabbbccc". Sugestion: utilisez la concaténation de chaînes et les index de chaînes.
2. Initialisez la chaîne "aaabbbccc" dans une variable nommée "s":



1. Utilisez une fonction qui vous permet de trouver la première occurrence de “b” dans la chaîne, et la première occurrence de “ccc”.
2. Utilisez une fonction qui vous permet de remplacer toutes les occurrences de “a” par “x”, et ensuite utilisez la même fonction pour remplacer seulement la première occurrence de “a” par “x”.
3. En commençant par la chaîne “aaa bbb ccc”, de quelles suites d’opérations avez-vous besoin pour arriver aux chaînes suivantes ? Vous pouvez utiliser la fonction “replace”.
  1. “AAA BBB CCC”
  2. “AAA bbb CCC”

# Chapter 3

## Les listes

Les listes de Python sont des structures de données qui regroupent des suites d'éléments. Les listes peuvent avoir des éléments de différents types et vous pouvez aussi mélanger des types différents au sein de la même liste, bien que la plupart du temps tous les éléments sont du même type.

Les listes sont créées en utilisant des crochets et les éléments sont séparés par des virgules. On peut accéder aux éléments d'une liste par leurs positions respectives. 0 est l'index du premier élément.

```
>>> l = [1, 2, 3, 4, 5]
>>> l[0]
1
>>> l[1]
2
```

Pouvez-vous accéder au numéro 4 de la liste précédente ?

Parfois vous voulez juste une petite portion d'une liste, une sous-liste. Les sous-listes peuvent être récupérées en utilisant une technique appelée *slicing*, qui consiste à définir les index de début et de fin.

```
>>> l = ['a', 'b', 'c', 'd', 'e']
>>> l[1:3]
['b', 'c']
```

Finalement, il est aussi possible de faire de l'arithmétique avec les listes, comme ajouter deux listes ensemble ou répéter the contenus d'une liste.

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2] * 2
[1, 2, 1, 2]
```

## Exercices avec des listes

Créez une liste nommée “l” avec les valeurs suivantes ([1, 4, 9, 10, 23]). En utilisant la documentation de Python sur les listes (<https://docs.python.org/3.5/tutorial/introduction.html#lists>) effectuez les exercices suivants:

1. En utilisant le slicing de liste obtenez les sous-listes [4, 9] et [10, 23].
2. Ajoutez (“append” en anglais) la valeur 90 à la fin de la liste “l”. Vérifiez la différence entre la concaténation et la méthode “append”.
3. Calculez la valeur moyenne de toutes les valeurs de la liste. Vous pouvez également utiliser les fonctions “sum” et “len”.
4. Supprimez la sous-liste [4, 9].

## Les compréhensions de liste

Une compréhension de liste est une manière concise d’écrire une liste. Elle consiste en des crochets contenant une expression suivie du mot-clé “for”. Le résultat sera une liste dont les résultats correspondent à l’expression. Voici comment créer une liste avec les nombres d’une autre liste, mais carrés.

```
>>> [x*x for x in [0, 1, 2, 3]]  
[0, 1, 4, 9]
```

Du fait de sa flexibilité, les compréhensions de liste utilisent généralement la fonction “range” qui retourne un intervalle (“range en anglais”):

```
>>> [x*x for x in range(4)]  
[0, 1, 4, 9]
```

Parfois vous pourriez vouloir filtrer les éléments selon une condition donnée. Le mot-clé “if” peut être utilisé dans ces cas-là:

```
>>> [x for x in range(10) if x % 2 == 0]  
[0, 2, 4, 6, 8]
```

L’exemple ci-dessus retourne toutes les valeurs paires dans un intervalle 0..10. Vous pouvez trouver plus d’informations sur les compréhensions de liste à <https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>.

## Exercices avec des compréhensions de liste

1. En utilisant des compréhensions de liste, créez une liste avec les carrés des dix premiers nombres.
2. En utilisant des compréhensions de liste, créez une liste avec les cubes des 20 premiers nombres.

3. Créez une compréhension de liste avec tous les nombres pairs de 0 à 20, et une autre avec tous les nombres impairs.
4. Créez une liste avec les carrés des nombres pairs de 0 à 20, et faites la somme des éléments de la liste en utilisant la fonction “sum”. Le résultat devrait être 1140. D’abord créez la liste en utilisant des compréhensions de listes, vérifiez le résultat, et ensuite appliquez la somme à la liste de compréhension.
5. Créez une compréhension de liste qui retourne une liste contenant les carrés de tous les nombres pairs de 0 à 20, mais ignore les nombres qui sont divisibles par 3. En d’autres mots, chaque nombre devrait être divisible par 2 et ne pas être divisible par 3. Recherchez le mot-clé “and” dans la documentation Python. La liste résultante est [4, 16, 64, 100, 196, 256]. # Modules et fonctions

Dans ce chapitre nous allons parler des modules et fonctions. Une fonction est un bloc de code qui est utilisé pour accomplir une action. Un module est un fichier Python contenant des variables, des fonctions et bien d’autres choses encore.

Lancez votre REPL Python. Utilisons le module “math” qui donne accès aux fonctions mathématiques:

```
>>> import math
>>> math.cos(0.0)
1.0
>>> math.radians(275)
4.799655442984406
```

Les fonctions sont des suites d’instructions qui sont exécutées lorsque la fonction est invoquée. Ce qui suit définit la fonction “do\_hello” qui imprime deux messages lorsqu’elle est invoquée:

```
>>> def do_hello():
...     print("Hello")
...     print("World")
...
>>> do_hello()
Hello
World
```

Assurez-vous de bien insérer un tab avant chacune des deux expressions dans la fonction précédente. Les tabs et les espaces ont une signification en Python et définissent qu’un bloc de code est d’une certaine manière dépendant de l’instruction précédente. Par exemple, les expressions print sont “à l’intérieur” de la fonction “do\_hello” et pour cette raison doivent avoir un tab.

Les fonctions peuvent également recevoir des paramètres et retourner une valeur (en utilisant le mot-clé “return”):

```
>>> def add_one(val):
...     print("Function got value", val)
...     return val + 1
...
>>> value = add_one(1)
```

```
Function got value 1
>>> value
2
```

## Exercices avec le module math

Utilisez la documentation Python concernant le module math (<https://docs.python.org/3/library/math.html>) pour résoudre les exercices suivants:

1. Trouvez le plus grand commun diviseur des paires de nombres suivantes: (15, 21), (152, 200), (1988, 9765).
2. Calculez le logarithme en base 2 des nombres suivants: 0, 1, 2, 6, 9, 15.
3. Utilisez la fonction “input” pour demander un nombre à l'utilisateur et montrez le résultat du sinus, du cosinus et de la tangente du nombre. Assurez-vous que vous convertissez l'input de l'utilisateur de chaîne à nombre (utilisez la fonction `int()` ou `float()`).

## Exercices avec les fonctions

1. Implémentez la fonction “add2” qui reçoit deux nombres comme arguments et retourne la somme des deux nombres. Ensuite implémentez la fonction “add3” qui reçoit et additionne 3 paramètres.
2. Implémentez une fonction qui retourne le plus grand de deux nombres donnés en paramètres. utilisez l'expression “if” pour comparer les deux nombres: <https://docs.python.org/3/tutorial/controlflow.html#if-statements>.
3. Implémentez une fonction nommée “is\_divisible” qui reçoit deux paramètres (nommés “a” et “b”) et retourne vrai si “a” peut être divisé par “b” ou faux autrement. Un nombre est divisible par un autre lorsque le reste de la division est zéro. Utilisez l'opérateur modulo (%).
4. Créez une fonction nommée “moyenne” qui calcule la valeur moyenne d'une liste passée en paramètre de la fonction. Utilisez les fonctions “sum” et “len”.

## Les fonctions récursives

En programmation, une fonction récursive est simplement une fonction qui s'appelle elle-même. Par exemple, prenez la fonction factorielle.

$$f(x) = \begin{cases} 1, & \text{if } x = 0. \\ x \times f(x - 1), & \text{otherwise.} \end{cases} \quad (3.1)$$

Prenons comme cas concret la factorielle de 5:

$$\begin{aligned} 5! &= 5 \times 4! \\ &= 5 \times 4 \times 3! \\ &= 5 \times 4 \times 3 \times 2! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned} \tag{3.2}$$

Pour l'essentiel, la factorielle de 5 est 5 fois la factorielle de 4, etc. Finalement, la factorielle de 1 (ou de zéro) est 1, ce qui rompt la récursion. En Python nous pourrions écrire la fonction récursive suivante:

```
def factorial(x):  
    if x == 0:  
        return 1  
    else:  
        return x * factorial(x-1)
```

L'astuce avec les fonctions récursives est qu'il doit y avoir un cas "de base" où la récursion s'arrête et un cas récursif qui itère vers le cas de base. Dans la cas de la factorielle nous savons que la factorielle de zero est un, et la factorielle d'un nombre plus grand que zero va dépendre de la factorielle du nombre précédent jusqu'à ce celui-ci atteigne zéro.

## Exercices avec les fonctions récursives

1. Implémentez la fonction factorielle et testez-là avec plusieurs valeurs. Vérifiez avec une calculatrice.
2. Implémentez une fonction récursive pour calculer la somme des (n) premiers nombres entiers (où (n) est un paramètre de fonction). Commencez par réfléchir au cas de base (la somme des 0 premiers entiers est égale à combien?) et ensuite réfléchissez au cas récursif.
3. La suite de Fibonacci est une suite de nombres dans laquelle chaque nombre de la suite correspond à la somme des deux nombres précédents. Étant donnée la définition récursive suivante, implémentez (fib(n)).

$$fib(n) = \begin{cases} 0, & \text{if } n = 0. \\ 1, & \text{if } n = 1. \\ fib(n-1) + fib(n-2), & \text{otherwise.} \end{cases} \tag{3.3}$$

Vérifiez vos résultats pour les premiers nombres de la suite: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... # L'itération et les boucles

Dans ce chapitre nous allons explorer les deux sujets de l'itération et des boucles. Les boucles sont utilisées en informatique pour automatiser les tâches répétitives.

En Python la forme la plus commune d’itération est la boucle “for”. La boucle “for” vous permet d’itérer sur tous les éléments d’une liste de manière à ce que vous puissiez faire tout ce que vous voulez avec chaque élément. Par exemple, créons une liste et imprimons la valeur du carré de chacun des éléments.

```
>>> for value in [0, 1, 2, 3, 4, 5]:
...     print(value * value)
...
0
1
4
9
16
25
```

C’est assez simple mais très puissant ! La boucle “for” est la base de beaucoup de choses en programmation. Par exemple, vous connaissez déjà la fonction “sum(list)” qui somme tous les éléments d’une liste, mais voici un exemple utilisant la boucle “for”:

```
>>> mylist = [1,5,7]
>>> sum = 0
>>> for value in mylist:
...     sum = sum + value
...
>>> print(sum)
13
```

Pour l’essentiel, vous créez la variable “sum” et vous continuez d’additionner chaque valeur selon l’ordre de la liste.

Parfois, au lieu des valeurs d’une liste, vous pourriez avoir besoin de travailler avec les index eux-mêmes, c’est-à-dire non pas avec les valeurs, mais avec les positions où elles sont dans la liste. Voici un exemple qui itère sur une liste et retourne les index et les valeurs pour chaque index:

```
>>> mylist = [1,5,7]
>>> for i in range(len(mylist)):
...     print("Index:", i, "Value:", mylist[i])
...
Index: 0 Value: 1
Index: 1 Value: 5
Index: 2 Value: 7
```

Vous pouvez voir que nous n’intérons pas sur la liste elle-même mais que nous itérons sur le “range” (l’intervalle) de la longueur de la liste. La fonction range retourne une liste spéciale:

```
>>> list(range(3))
[0, 1, 2]
```

Donc, quand vous utilisez “range” vous n’itérez pas sur “myList” mais sur une liste contenant plusieurs nombres que vous allez utiliser comme index pour accéder aux valeurs

individuelles de “myList”. Vous trouverez plus d’informations sur la fonction range dans la documentation Python à <https://docs.python.org/3/tutorial/controlflow.html#the-range-function>.

Il se peut que parfois vous ayez besoin des deux, index et valeurs. Vous pouvez alors utiliser la fonction “enumerate”:

```
>>> mylist = [1,5,7]
>>> for i, value in enumerate(mylist):
...     print("Index:", i, "Value:", value)
...
Index: 0 Value: 1
Index: 1 Value: 5
Index: 2 Value: 7
```

Souvenez-vous que la première valeur d’une liste Python est toujours à l’index 0.

Finalement, nous avons aussi l’expression “while” qui nous permet de répéter une suite d’instructions tant qu’une condition spécifiée est vraie. Par exemple, l’exemple suivant démarre avec “n” à 10 et **tant que “n” est plus grand que 0**, il continue de soustraire 1 de “n”. Lorsque “n” atteint 0, la condition “n > 0” est fausse, et la boucle s’arrête.

```
>>> n = 10
>>> while n > 0:
...     print(n)
...     n = n-1
...
10
9
8
7
6
5
4
3
2
1
```

Remarquez qu’il n’imprime jamais 0...

## Exercices avec la boucle for

Pour cette section, il se pourrait que vous souhaitiez consulter la documentation Python à <https://docs.python.org/3/tutorial/controlflow.html#the-range-function>.

1. Créez une fonction “ajouter” qui reçoit une liste comme paramètre et retourne la somme de tous les éléments de la liste. Utilisez la boucle “for” pour itérer à travers les éléments de la liste.
2. Créez une fonction qui reçoit une liste comme paramètre et retourne la valeur



maximum de la liste. Lorsque vous itérez à travers la liste vous voudrez garder la valeur maximum trouvée jusque-là afin de la comparer avec les éléments suivants de la liste.

3. Modifiez la fonction précédente de manière qu'elle retourne une liste dont le premier élément est la valeur maximum et le second l'index de la valeur maximum de la liste. En plus de conserver la valeur maximum de la liste à chaque étape de la boucle, vous devrez conserver aussi la position où elle est apparue.
4. Implémentez une fonction qui retourne en ordre inverse une liste reçue en paramètre. Vous pouvez créer une liste vide et ajouter les valeurs dans le sens inverse au fur et à mesure qu'elles sortent de la liste d'origine. Regardez ce que vous pouvez faire avec les listes à <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>.
5. Créez une fonction "is\_sorted" qui reçoit une liste comme paramètre et retourne True si la liste est triée dans un ordre ascendant. Par exemple [1, 2, 2, 3] est triée alors que [1, 2, 3, 2] ne l'est pas. Suggestion: Vous devez comparer un nombre dans la liste avec le suivant. Vous pouvez utiliser les index, ou vous devez garder en mémoire le nombre précédent dans une variable pendant que vous itérez à travers la liste.
6. Implémentez la fonction "is\_sorted\_dec" qui est similaire à la précédente mais cette fois-ci tous les éléments doivent être triés en ordre décroissant.
7. Implémentez la fonction "has\_duplicates" qui vérifie si une liste a des valeurs dupliquées. Il se peut que vous ayez à utiliser deux boucles "for", dans lesquelles pour chaque valeur vous ayez à chercher des doublons dans le reste de la liste.

## Exercices avec l'expression while

1. Implémentez une fonction qui reçoit un nombre comme paramètre et imprime, en ordre décroissant, quels nombres sont pairs et quels nombres sont impairs, jusqu'à atteindre 0.

```
>>> even_odd(10)
Even number: 10
Odd number: 9
Even number: 8
Odd number: 7
Even number: 6
Odd number: 5
Even number: 4
Odd number: 3
Even number: 2
Odd number: 1
```

# Chapter 4

## Les Dictionnaires

Dans ce chapitre nous allons travailler avec les dictionnaires Python. Les dictionnaires sont des structures de données qui indexent les valeurs par une clé donnée (paires clé-valeur). L'exemple suivant montre un dictionnaire qui indexe les âges des étudiants par nom.

```
ages = {  
    "Peter": 10,  
    "Isabel": 11,  
    "Anna": 9,  
    "Thomas": 10,  
    "Bob": 10,  
    "Joseph": 11,  
    "Maria": 12,  
    "Gabriel": 10,  
}  
  
>>> print(ages["Peter"])  
10
```

Il est possible d'itérer sur les contenus d'un dictionnaire en utilisant "items", comme ceci:

```
>>> for name, age in ages.items():  
...     print(name, age)  
...  
Peter 10  
Isabel 11  
Anna 9  
Thomas 10  
Bob 10  
Joseph 11  
Maria 12  
Gabriel 10
```

Néanmoins, les clés de dictionnaire ne doivent pas être nécessairement des chaînes de caractères mais peuvent être n'importe quel objet immuable:

```
d = {
    0: [0, 0, 0],
    1: [1, 1, 1],
    2: [2, 2, 2],
}

>>> d[2]
[2, 2, 2]
```

Et vous pouvez aussi utiliser d'autres dictionnaires comme valeurs:

```
students = {
    "Peter": {"age": 10, "address": "Lisbon"},
    "Isabel": {"age": 11, "address": "Sesimbra"},
    "Anna": {"age": 9, "address": "Lisbon"},
}

>>> students['Peter']
{'age': 10, 'address': 'Lisbon'}
>>> students['Peter']['address']
'Lisbon'
```

C'est très utile pour structurer une information hiérarchique.

## Exercices avec les dictionnaires

Utilisez la documentation Python à <https://docs.python.org/3/library/stdtypes.html#mapping-types-dict> pour résoudre les exercices suivants.

Prenez le dictionnaire Python suivant:

```
ages = {
    "Peter": 10,
    "Isabel": 11,
    "Anna": 9,
    "Thomas": 10,
    "Bob": 10,
    "Joseph": 11,
    "Maria": 12,
    "Gabriel": 10,
}
```

1. Combien d'étudiants sont dans le dictionnaire? Etudiez la fonction "len".
2. Implémentez une fonction qui reçoit le dictionnaire "ages" comme paramètre et retourne l'âge moyen des étudiants. Traversez tous les éléments du dictionnaire en utilisant la méthode "items" comme expliqué plus haut.
3. Implémentez une fonction qui reçoit le dictionnaire "ages" comme paramètre et retourne le nom de l'étudiant le plus âgé.

4. Implémentez une fonction qui reçoit le dictionnaire “ages” et une nombre “n” et retourne un nouveau dictionnaire dans lequel chaque étudiant est plus âgé de (n) ans. Par exemple, `new_ages(ages, 10)` retourne une copie de “ages” dans laquelle chaque étudiant est 10 ans plus âgé.

## Exercices avec des sous-dictionnaires

Prenez le dictionnaire suivant:

```
students = {
    "Peter": {"age": 10, "address": "Lisbon"},
    "Isabel": {"age": 11, "address": "Sesimbra"},
    "Anna": {"age": 9, "address": "Lisbon"},
}
```

1. Combien d’étudiants sont-ils dans le dict “students”? Utilisez la fonction appropriée.
2. Implémentez une fonction qui reçoit le dict “students” et retourne l’âge moyen.
3. Implémentez une fonction qui reçoit le dict “students” et une adresse, et retourne une liste avec les noms de tous les étudiants dont l’adresse correspond à l’adresse dans l’argument. Par exemple, invoquer “`find_students(students, 'Lisbon')`” devrait retourner Peter et Anna. # Les Classes

En programmation orientée objet (POO), une classe est une structure qui permet de grouper ensemble un ensemble de propriétés (appelées attributs) et de fonctions (appelées méthodes) pour manipuler ces propriétés. Prenez la classe suivante qui définit une personne avec les propriétés “name” et “age” et la méthode “greet”.

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, my name is %s!" % self.name)
```

La plupart des classes ont besoin d’une méthode appelée constructeur (“\_\_init\_\_”) pour initialiser les attributs de la classe. Dans le cas précédent le constructeur de la classe reçoit le nom et l’âge de la personne et stocke cette information dans l’instance de la classe (référéncée par le mot-clé `self`). Finalement, la méthode “greet” imprime le nom de la personne tel que stocké dans une instance de classe particulière (un objet).

Les instances de classe sont utilisées grâce à une instanciation d’objets. Voici comment nousinstancions deux objets:

```
>>> a = Person("Peter", 20)
>>> b = Person("Anna", 19)

>>> a.greet()
```

```

Hello, my name is Peter!
>>> b.greet()
Hello, my name is Anna!

>>> print(a.age)  # Nous pouvons aussi accéder aux attributs d'un objet.
20

```

## Exercices avec des classes

Utilisez la documentation Python sur les classes à <https://docs.python.org/3/tutorial/classes.html> pour résoudre les exercices suivants.

1. Implémentez une classe nommée “Rectangle” pour stocker les coordonnées d’un rectangle d’après le coin supérieur gauche (x1, y1) et le coin inférieur droit (x2, y2).
2. Implémentez le constructeur de classe avec les paramètres (x1, y1, x2, y2) et stockez-les dans l’instance de classe en utilisant le mot-clé “self”.
3. Implémentez les méthodes “width()” et “height()” qui retournent, respectivement, la largeur et la hauteur d’un rectangle. Créez deux objets, instances de “Rectangle”, pour tester les calculs.
4. Implémentez la méthode “area” pour retourner l’aire du rectangle (width\*height).
5. Implémentez la méthode “circumference” qui retourne le périmètre du rectangle (2\*width + 2\*height).
6. Faites un print d’un des objets créés pour tester la classe. Implémentez la méthode “\_\_str\_\_” de manière que lorsque vous imprimez l’un des objets il imprime les coordonnées comme (x1, y1)(x2, y2).

## L’Héritage de classe

En programmation orientée objet, l’héritage est une des formes par lesquelles une sous-classe peut hériter des attributs et des méthodes d’une autre classe, lui permettant de réécrire certaines des fonctionnalités de la superclasse. Par exemple, à partir de la classe “Person” vue plus haut nous pouvons créer une sous-classe qui ne conserve que les personnes âgées de 10 ans:

```

class TenYearOldPerson(Person):

    def __init__(self, name):
        super().__init__(name, 10)

    def greet(self):
        print("I don't talk to strangers!!")

```

L'indication que la classe "TenYearOldPerson" est une sous-classe de "Person" est donnée à la première ligne. Ensuite, nous avons réécrit le constructeur de la sous-classe pour ne recevoir que le nom de la personne, mais nous appelons finalement le constructeur de la superclasse avec le nom de l'enfant de 10 ans et l'âge codé en dur de 10. Enfin, nous avons réimplémenté la méthode "greet".

## Exercices avec l'héritage

Utilisez la classe "Rectangle" telle qu'elle est implémentée plus haut pour effectuer les exercices suivants:

1. Créez une class "Square" comme sous-classe de "Rectangle".
  2. Implement the "Square" constructor. The constructor should have only the x1, y1 coordinates and the size of the square. Notice which arguments you'll have to use when you invoke the "Rectangle" constructor when you use "super".
  3. Implémentez le constructeur de "Square". Le constructeur ne doit contenir que les coordonnées x1, y1 et la taille du carré. Faites attention aux arguments que vous utilisez lorsque vous invoquez le constructeur de "Rectangle" lorsque vous utilisez "super".
  4. Instantiez deux objets "Square", invoquez la méthode "area" et imprimez les objets. Assurez-vous que tous les calculs retournent des nombres corrects et que les coordonnées des carrés sont cohérents avec la taille du carré utilisée comme argument.
- # Les Itérateurs

Comme nous l'avons vu précédemment, en Python nous utilisons la boucle "for" pour itérer sur les contenus des objets:

```
>>> for value in [0, 1, 2, 3, 4, 5]:
...     print(value)
...
0
1
4
9
16
25
```

Les objets qui peuvent être utilisés dans une boucle "for" sont appelés itérateurs. Partant de là, un itérateur est un objet qui suit le protocole d'itération.

La fonction intégrée "iter" peut-être utilisée pour construire des objets intérateurs, tandis que la fonction "next" peut être utilisée pour itérer graduellement sur leur contenu:

```
>>> my_iter = iter([1, 2, 3])
>>> my_iter
<list_iterator object at 0x10ed41cc0>
>>> next(my_iter)
1
```

```

>>> next(my_iter)
2
>>> next(my_iter)
3
>>> next(my_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

S'il n'y a plus d'élément, l'itérateur lève une exception "StopIteration".

## Les Classes Itérateurs

Les itérateurs peuvent être implémentés comme classes. Vous devez seulement implémenter les méthodes "`__next__`" et "`__iter__`". Voici un exemple de classe qui imite la fonction "range", retournant toutes les valeurs de "a" jusqu'à "b":

```

class MyRange:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __iter__(self):
        return self

    def __next__(self):
        if self.a < self.b:
            value = self.a
            self.a += 1
            return value
        else:
            raise StopIteration

```

Pour l'essentiel, à chaque appel, la méthode "next" augmente la variable "a" d'une unité et retourne sa valeur. Lorsque celle-ci atteint la valeur de "b", la méthode lève une exception StopIteration.

```

>>> myrange = MyRange(1, 4)
>>> next(myrange)
1
>>> next(myrange)
2
>>> next(myrange)
3
>>> next(myrange)
Traceback (most recent call last):

```

```
File "<stdin>", line 1, in <module>
StopIteration
```

Mais le plus important, c'est que vous pouvez utiliser la classe itérateur dans une boucle "for":

```
>>> for value in MyRange(1, 4):
...     print(value)
...
1
2
3
```

## Exercices avec les itérateurs

1. Implémentez une classe itérateur qui retourne le carré de tous les nombres de "a" à "b".
2. Implémentez une classe itérateur qui retourne tous les nombres pairs de 1 à (n).
3. Implémentez une classe itérateur qui retourne tous les nombres impairs de 1 à (n).
4. Implémentez une classe itérateur qui retourne tous les nombres de (n) positif jusqu'à 0.
5. Implémentez une classe itérateur qui retourne la suite de Fibonnaci depuis le premier élément jusqu'à (n). Vous pouvez vérifier la définition de la suite de Fibonnaci dans le chapitre sur les fonctions. Voici les premiers nombres de la suite: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
6. Implémentez une classe itérateur qui retourne toutes les paires consécutives de nombres de 0 à (n), telles que (0, 1), (1, 2), (2, 3)... # Les Générateurs

Si vous avez lu le chapitre précédent, vous savez que les itérateurs sont des objets qui sont régulièrement utilisés dans les boucles "for". En d'autres mots, les itérateurs sont des objets qui implémentent le protocole d'itération. Un générateur Python est un moyen pratique d'implémenter un itérateur. Au lieu d'une classe, un générateur est une fonction qui retourne une valeur à chaque fois que le mot-clé "yield" est utilisé. Voici l'exemple d'un générateur qui compte les valeurs entre deux nombres:

```
def myrange(a, b):
    while a < b:
        yield a
        a += 1
```

Tout comme les itérateurs, les générateurs peuvent être utilisés dans une boucle "for":

```
>>> for value in myrange(1, 4):
...     print(value)
...
1
```



```
2
3
```

Sous le capot, les générateurs se comportent d’une manière semblable aux itérateurs:

```
>>> seq = myrange(1,3)
>>> next(seq)
1
>>> next(seq)
2
>>> next(seq)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La chose intéressante à propos des générateurs est le mot-clé “yield”. Le mot-clé “yield” fonctionne de manière similaire au mot-clé “return”, mais à la différence de “return”, il permet à la fonction de reprendre son exécution. En d’autres mots, à chaque fois que la valeur suivante d’un générateur est demandée, Python réveille la fonction et reprend son exécution depuis la ligne commençant par “yield” comme si la fonction ne s’était jamais interrompue.

Les fonctions générateurs peuvent utiliser d’autres fonctions à l’intérieur d’elles-mêmes. Par exemple, il est très courant d’utiliser la fonction “range” pour itérer sur une suite de nombres.

```
def squares(n):
    for value in range(n):
        yield value * value
```

## Exercices avec les générateurs

1. Implémentez un générateur appelé “squares” qui produit (yield) les carrés de tous les nombres de (a) à (b). Testez-la avec une boucle “for” et imprimez chacune des valeurs produites.
2. Créez un générateur qui produit tous les nombres pairs de 1 à (n).
3. Créez un autre générateur qui produit tous les nombres impairs de 1 à (n).
4. Implémentez un générateur qui renvoie tous les nombres de (n) positif jusqu’à 0.
5. Créez un générateur qui renvoie la suite de Fibonacci en commençant par le premier élément jusqu’à (n). Les premiers nombres de la suite sont: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
6. Implémentez un générateur qui renvoie toutes les paires consécutives de nombres de 0 à (n),, sous la forme (0, 1), (1, 2), (2, 3)... # Les Coroutines

Les coroutines de Python sont semblables aux générateurs, mais à la place de produire des données, les coroutines sont surtout utilisées pour consommer des données. En d’autres

mots, les coroutines sont des fonctions qui reprennent leur activité chaque fois qu’une valeur est envoyée en utilisant la méthode `send`.

La particularité des coroutines est leur usage du mot-clé `yield` du côté droit d’une expression d’assignation. Voici un exemple d’une coroutine qui imprime les valeurs qui lui sont envoyées:

```
def coroutine():
    print('My coroutine')
    while True:
        val = yield
        print('Got', val)

>>> co = coroutine()
>>> next(co)
My coroutine
>>> co.send(1)
Got 1
>>> co.send(2)
Got 2
>>> co.send(3)
Got 3
```

L’appel initial à `next` est requis pour faire démarrer la coroutine. Vous pouvez voir qu’elle exécute l’expression `print`. Lorsque la fonction atteint l’expression `yield`, elle va attendre d’être relancée. Ensuite, à chaque fois qu’une valeur est envoyée (avec `send`), la fonction coroutine reprend depuis le `yield`, copie la valeur dans `val` et l’imprime.

Les coroutines peuvent être refermées avec la méthode `close()`.

```
>>> co.close()
>>> co.send(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

## Exercices avec les coroutines

1. Créez une coroutine nommée “square” qui imprime le carré de toute valeur qui lui est envoyée.
2. Implémentez la coroutine “minimize” qui retient et imprime la valeur minimale envoyée à la fonction.

## Les Pipelines

Les coroutines peuvent être utilisées pour implémenter des pipelines de données dans lesquelles une coroutine va envoyer des données à la coroutine suivante dans le pipeline.

Les couroutines insèrent des données dans le pipeline en utilisant la méthode `send()`.



Figure 4.1:

Voici un exemple d’une petites pipeline dans laquelle les valeurs envoyées à la coroutine “producer” sont mises au carré et envoyée à la coroutine “consumer” pour être imprimées:

```
def producer(consumer):
    print("Producer ready")
    while True:
        val = yield
        consumer.send(val * val)

def consumer():
    print("Consumer ready")
    while True:
        val = yield
        print('Consumer got', val)
```

Comme plus haut, les coroutines doivent être “initialisées” avec `next` avant qu’aucune valeur ne puisse être envoyée.

```
>>> cons = consumer()
>>> prod = producer(cons)
>>> next(prod)
Producer ready
>>> next(cons)
Consumer ready

>>> prod.send(1)
Consumer got 1
>>> prod.send(2)
Consumer got 4
>>> prod.send(3)
Consumer got 9
```

Aussi, avec les coroutines, les données peuvent être envoyées vers de multiple destinations. L’exemple suivant implémente deux consommateurs dont le premier imprime seulement les nombres compris entre 0 et 10 et le second imprime seulement les nombre de 10 à 20:

```

def producer(consumers):
    print("Producer ready")
    try:
        while True:
            val = yield
            for consumer in consumers:
                consumer.send(val * val)
    except GeneratorExit:
        for consumer in consumers:
            consumer.close()

def consumer(name, low, high):
    print("%s ready" % name)
    try:
        while True:
            val = yield
            if low < val < high:
                print('%s got' % name, val)
    except GeneratorExit:
        print("%s closed" % name)

```

Comme plus haut, les coroutines doivent être “initialisées” avant qu’une valeur puisse être envoyée.

```

>>> con1 = consumer('Consumer 1', 00, 10)
>>> con2 = consumer('Consumer 2', 10, 20)
>>> prod = producer([con1, con2])

>>> next(prod)
Producer ready
>>> next(con1)
Consumer 1 ready
>>> next(con2)
Consumer 2 ready

>>> prod.send(1)
Consumer 1 got 1
>>> prod.send(2)
Consumer 1 got 4
>>> prod.send(3)
Consumer 1 got 9
>>> prod.send(4)
Consumer 2 got 16

>>> prod.close()
Consumer 1 closed
Consumer 2 closed

```

Les données sont envoyées à tous les consommateurs, mais seulement le second exécute

l'expression `print`. Remarquez l'utilisation de l'exception `GeneratorExit`. Il peut parfois être utile d'intercepter l'exception et d'informer les coroutines en aval que la pipeline n'est plus utile.

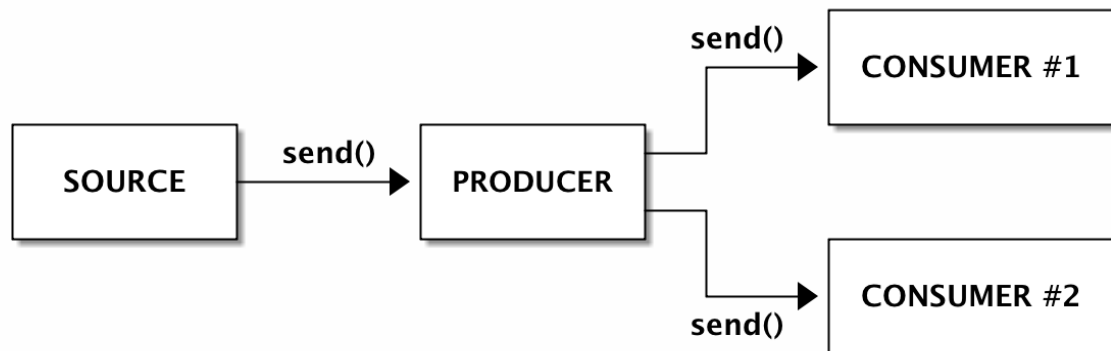


Figure 4.2:

## Exercices avec des pipelines de coroutines

1. Implémentez une pipeline producteur-consommateur dans laquelle toutes les valeurs mises au carré par le producteur sont envoyées à deux consommateurs. L'un devrait stocker et imprimer la valeur minimum jusqu'à là, et l'autre la valeur maximum.
2. Implémentez une pipeline producteur-consommateur dans laquelle les valeurs mises au carré par le producteur sont envoyées à deux consommateurs, un à la fois. La première valeur devrait être envoyée au consommateur 1, la seconde valeur au consommateur 2, la troisième valeur à nouveau au consommateur 1, et ainsi de suite. Fermer le producteur devrait forcer les consommateurs à imprimer une liste contenant les nombres que chacun d'eux a obtenu. # La Programmation Asynchrone

Jusqu'ici nous avons fait de la *programmation synchrone*. L'exécution d'un programme synchrone est assez simple: le programme démarre à la première ligne, et ensuite chaque ligne est exécutée jusqu'à ce que le programme atteigne sa fin. Chaque fois qu'une fonction est appelée, le programme attend que la fonction retourne avant de continuer à la ligne suivante.

En programmation asynchrone, l'exécution d'une fonction est habituellement non-bloquante. En d'autres termes, chaque fois que vous appelez une fonction celle-ci retourne immédiatement. Néanmoins cette fonction n'est pas nécessairement exécutée tout de suite. Il y a généralement un mécanisme (appelé le "scheduler") qui est responsable de l'exécution future de la fonction.

Le problème avec la programmation asynchrone est qu'un programme peut s'arrêter avant le démarrage de la moindre fonction asynchrone. Une solution habituelle à ce problème est que les fonction asynchrones retournent des "futures" ou "promises". Ce sont des objets

qui représentent l'état d'exécution d'une fonction asynchrone. Enfin, les frameworks de programmation asynchrone ont typiquement des mécanismes pour bloquer ou attendre que ces fonctions asynchrones finissent leur travail, mécanismes basés sur ces objets "futures".

Depuis Python 3.6, le module "asyncio" combiné avec les mots-clés *async* et *await* nous permet d'implémenter ce qu'on appelle des *programmes multitâches coopératifs*. Dans ce type de programmation, une fonction coroutine cède le contrôle volontairement à une autre fonction coroutine lorsqu'elle est inactive ou qu'elle attend de nouvelles données.

Considérez la fonction asynchrone suivante qui met au carré un nombre et dort une seconde avant de retourner. Les fonctions asynchrones sont déclarées avec **async def**. Ignorez le mot-clé **await** pour l'instant:

```
import asyncio

async def square(x):
    print('Square', x)
    await asyncio.sleep(1)
    print('End square', x)
    return x * x

# Créez une boucle d'événements
loop = asyncio.get_event_loop()

# Exécutez une fonction asynchrone et attendez qu'elle arrive à son terme

results = loop.run_until_complete(square(1))
print(results)

# Fermez la boucle
loop.close()
```

La boucle d'événements (<https://docs.python.org/3/library/asyncio-eventloop.html>) est, entre autres, le mécanisme Python qui organise l'exécution des fonctions asynchrones. Nous utilisons la boucle pour faire tourner la fonction jusqu'à complétion. Elle est le mécanisme de synchronisation qui s'assure que l'expression `print` suivante ne soit pas exécutée tant que nous avons des résultats.

L'exemple précédent n'est pas un bon exemple de programmation asynchrone parce que nous n'avons pas besoin de tant de complexité pour n'exécuter qu'une seule fonction. Néanmoins, imaginez que vous ayez besoin d'exécuter la fonction `square(x)` trois fois, comme ceci:

```
square(1)
square(2)
square(3)
```

Since the `square()` function has a sleep function inside, the total execution time of this program would be 3 seconds. However, given that the computer is going to be idle for a full second each time the function is executed, why can't we start the next call while the previous is sleeping? Here's how we do it:

Puisque la fonction `square()` contient une fonction `sleep`, l'exécution totale de ce programme devrait prendre 3 secondes. Néanmoins, puisque l'ordinateur va rester inactif pendant une seconde complète chaque fois que la fonction est exécutée, pourquoi ne pas démarrer l'appel suivant pendant que le précédent dort ? Voici comment faire:

```
# Exécutez la fonction async et attendez qu'elle se termine
results = loop.run_until_complete(asyncio.gather(
    square(1),
    square(2),
    square(3)
))
print(results)
```

Pour l'essentiel, nous utilisons `asyncio.gather(*tasks)` pour informer la boucle d'attendre que toutes les tâches soient finies. Puisque les coroutines démarreront à peu près au même moment, le programme devrait tourner seulement 1 seconde. La fonction `gather()` du module `Asyncio` n'exécutera pas nécessairement les coroutines dans l'ordre, bien qu'elle retournera une liste de résultats.

```
$ python3 python_async.py
Square 2
Square 1
Square 3
End square 2
End square 1
End square 3
[1, 4, 9]
```

Parfois on peut avoir besoin de résultats aussitôt qu'ils sont disponibles. Pour cela nous pouvons utiliser une seconde coroutine qui gère chacun des résultats en utilisant `asyncio.as_completed()`:

```
(...)

async def when_done(tasks):
    for res in asyncio.as_completed(tasks):
        print('Result:', await res)

loop = asyncio.get_event_loop()
loop.run_until_complete(when_done([
    square(1),
    square(2),
    square(3)
]))
```

Cela imprimera quelque chose comme:

```
Square 2
Square 3
Square 1
End square 3
```

```
Result: 9
End square 1
Result: 1
End square 2
Result: 4
```

Finalement, les coroutines async peuvent appeler **d'autres fonctions coroutines asynchrones** grâce au mot-clé **await**:

```
async def compute_square(x):
    await asyncio.sleep(1)
    return x * x

async def square(x):
    print('Square', x)
    res = await compute_square(x)
    print('End square', x)
    return res
```

## Exercices avec asyncio

1. Implémentez une fonction coroutine asynchrone qui additionne deux variables et dort un nombre de secondes égal au résultat de l'addition. Utilisez la boucle asyncio pour appeler la fonction avec deux nombres.
2. Changez le programme précédent pour organiser l'exécution de deux appels à la fonction somme.