Informatique - Support d'enseignement 1re année de licence, Univ. Lille

Philippe Marquet, Maude Pupin

▶ **To cite this version:**

Philippe Marquet, Maude Pupin. Informatique - Support d'enseignement 1re année de licence, Univ. Lille. Licence. Informatique, Université de Lille, Villeneuve d'Ascq, France. 2022, pp.211. hal-04131704

HAL Id: hal-04131704
https://hal.science/hal-04131704

Submitted on 16 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Informatique — Support d'enseignement[1]

1re année de licence, Univ. Lille

Équipe pédagogique, Formation en informatique de Lille
Faculté des sciences et technologies, Université de Lille

septembre-décembre 2020

[1]Philippe Marquet, Maude Pupin, avec le soutien de Laurent Noé, Pierre Allegraud, Patrice Thibaud, inspiré·es des riches versions antérieures de l'Équipe pédagogique.
Document mis à disposition selon les termes de la licence CC BY — Attribution 4.0 International.
Contact : philippe.marquet@univ-lille.fr

Ce support d'enseignement a été rédigé entre septembre et décembre 2022 à l'occasion de la refonte de l'enseignement d'*Informatique* de 1re année de licence proposé dans les portails SESI – Sciences exactes et sciences de l'ingénierie –, MIASHS – Mathématiques et informatique appliquées aux sciences humaines et sociales –, et PEIP – Parcours des écoles d'ingénieurs Polytech – de la Faculté des sciences et technologies à l'Université de Lille.

Bien entendu la rédaction de ces supports d'enseignement s'est appuyée sur les riches versions antérieures produites par les équipes pédagogiques successives du département, FIL – Formations en informatique de Lille – que nous remercions chaleureusement.

Ont participé à la rédaction de ce document : Philippe Marquet, Maude Pupin, avec le soutien de Laurent Noé, Pierre Allegraud, et Patrice Thibaud.

Ce document est mis à disposition selon les termes de la Licence Creative Commons Attribution 4.0 International (CC BY 4.0), `creativecommons.org/licenses/by/4.0/deed.fr`

Table des matières

1. Valeurs, types et variables — Expressions, instructions . . . 4
Valeur et type – chaînes de caractères . . . 5
Variables . . . 8
Nombres entiers . . . 11
Nombres à virgule flottante . . . 13
Expressions et instructions . . . 16
Exercices niveau débutant . . . 19
Exercices niveau intermédiaire . . . 21
2. Fonctions . . . 23
Utilisation de fonctions . . . 24
Bibliothèque et importation de fonctions . . . 27
Définition de fonction . . . 29
Test de fonction . . . 33
Test de fonction avec L1test . . . 37
Exercices niveau débutant . . . 42
Exercices niveau intermédiaire . . . 44
Exercices niveau confirmé . . . 47
3. Type booléen - Expressions booléennes - Prédicats . . . 50
Expressions booléennes . . . 51
Opérateurs logiques. . . . 56
Expressions booléennes et doctests . . . 60
Expressions booléennes et L1test . . . 63
Exercices niveau débutant . . . 67
Exercices niveau intermédiaire . . . 69
Exercices niveau confirmé . . . 72
4. Instruction conditionnelle . . . 75
Instruction conditionnelle . . . 76
Instruction conditionnelle et alternative . . . 79
Instruction conditionnelle - Alternatives multiples . . . 82
Exercices niveau débutant . . . 85
Exercices niveau intermédiaire . . . 88
Exercices niveau confirmé . . . 93
5. Interaction avec l'utilisateur . . . 96
Interaction avec l'utilisateur — Programme . . . 97
Écrire sur le terminal . . . 99
Lire depuis le terminal. . . . 102
Exercices niveau débutant . . . 105
Exercices niveau intermédiaire . . . 108
6. Boucles - for . . . 112
Itérable et boucle 'for' . . . 113
Intervalle 'range' . . . 116
Utilisations classiques de boucles 'for' . . . 118
Boucles 'for' imbriquées . . . 122
Exercices niveau débutant . . . 124
Exercices niveau intermédiaire . . . 128
Exercices niveau confirmé . . . 130
Exercices complémentaires . . . 132

7. Chaînes de caractères 137
Rappels sur les chaînes de caractères 138
Indices pour les chaînes de caractères 140
Codage des caractères et comparaison des chaînes de caractères 143
Non mutabilité des chaînes de caractères 147
Exercices niveau débutant 151
Exercices niveau intermédiaire 153
Exercices niveau confirmé 156
8. Itération conditionnelle - while 158
Itération conditionnelle 159
Utilisations typiques de boucles ‘while‘ 163
Autres utilisations typiques de boucles ‘while‘ 167
Exercices niveau débutant 170
Exercices niveau intermédiaire 173
Exercices niveau confirmé 177
9. Listes Python - 1re partie 178
Découvrir les listes Python 179
Modifier une liste Python 183
Exercices niveau débutant 185
Exercices niveau intermédiaire 188
Exercices niveau confirmé 191
Exercices complémentaires 193
10. Listes Python - 2de partie 194
Aliasing de listes, valeurs mutables 195
Copier une liste 200
Copier ou muter une liste ? 203
Exercices niveau débutant 207

1. Valeurs, types et variables — Expressions, instructions

Valeur et type – chaînes de caractères

Coup d'œil

Premier support de cours. On découvre

- les notions de type et valeur en informatique
- les chaînes de caractères

Types de valeurs

L'informatique est d'un certain point de la vue la science de la manipulation d'informations.

Les informations sont représentées par des **valeurs**.

En Python, toutes les valeurs possèdent un **type**.

Le type d'une valeur définit :

- l'ensemble des valeurs qui peuvent être prises
- les **opérations** qu'il est possible de lui appliquer.

Les types de base sont :

- les entiers, appelés `int`
- les nombres à virgules flottantes (approximation des réels), appelés `float`
- les booléens, appelés `bool`
- les chaînes de caractères, appelées `str`.

Dans un premier temps, nous allons découvrir les chaînes de caractères.
Puis nous nous intéresserons aux nombres.
Enfin, nous traiterons des booléens lors de prochaines séances.

Chaîne de caractères

Un **caractère** est un symbole qui peut être écrit :

- une lettre (de l'alphabet latin) : `a`, `b`, etc., `X`, `Y`, `Z`
- une lettre accentuée : `à`, `û`, `É`, etc.
- un chiffre : `0`, `1`, etc.
- un symbole de ponctuation tel le point `.`, le point-virgule `;`, etc.
- une espace
- une lettre d'autres alphabets (non-latin) : α, Φ, ت, ...

Une **chaîne de caractères** est une séquence de caractères, c'est–à–dire des caractères qui se suivent les uns derrières les autres.

En anglais, on utilise le terme *string*.

Le nombre de caractères d'une chaîne de caractères n'est pas contraint.

Une chaîne de caractères peut ne contenir aucun caractère : on l'appelle alors **chaîne vide**.

Valeur littérale d'une chaîne de caractères

En informatique, les **valeurs littérales** sont les manières d'écrire des valeurs.

Pour les chaînes de caractères, on écrit tout simplement la séquence des caractères entre deux délimiteurs. On utilise habituellement l'apostrophe `'` comme délimiteur.

On écrit donc par exemple :

```
>>> 'Noémie programme en Python'
'Noémie programme en Python'
```

ou

```
>>> '314'
'314'
```

La chaîne vide peut s'écrire :

```
>>> ''
''
```

On peut aussi utiliser les guillemets `"`, en particulier quand la chaîne de caractères comporte une apostrophe :

```
>>> "Noémie s'amuse avec Python"
"Noémie s'amuse avec Python"
```

Le type `str`

On peut interroger l'interpréteur Python sur le type d'une valeur :

```
>>> type('Noémie programme en Python')
<class 'str'>
>>> type("Noémie s'amuse avec Python")
<class 'str'>
```

Autre exemple avec la chaîne vide :

```
>>> type('')
<class 'str'>
```

ou pour une chaîne constituée exclusivement de chiffres :

```
>>> type('314')
<class 'str'>
```

Opérations sur les chaînes de caractères

Des opérations sont définies sur les valeurs de type chaîne de caractères. Nous en étudions une première ici.

La **concaténation** de deux chaînes de caractères :

- produit une chaîne de caractères
- dont la séquence de caractères est la mise bout à bout des deux chaînes de caractères.

Cette opération est notée `+`.

On peut donc écrire :

```
>>> 'Noémie ' + 'programme en Python'
'Noémie programme en Python'
>>> 'Noémie ' + "s'amuse avec Python"
"Noémie s'amuse avec Python"
```

ou encore :

```
>>> '31' + '4'
'314'
```

Memento

- un type définit l'ensemble des valeurs valides pour ce type
- le type d'une valeur définit l'ensemble des opérations possibles sur ces valeurs
- une valeur littérale est une manière d'écrire des valeurs
- une chaîne de caractères est une séquence de caractères
- on écrit les caractères d'une chaîne entre délimiteurs (apostrophes ou guillemets)
- l'opération de concaténation de deux chaînes de caractères est notée +
- en Python, on peut connaître le type d'une valeur avec `type(…)`

Variables

Coup d'œil

On découvre :

- la notion de variable en informatique
- l'instruction d'affectation

Variables et affectation

Une variable est un nom qui va permettre de désigner une valeur.

Ce nom sera remplacé par la valeur associée à la variable.

L'opération d'**affectation** permet de lier une valeur à la variable.

La syntaxe de l'affectation (c'est-à-dire la manière d'écrire une affectation) est illustrée par l'exemple suivant

```
>>> prenom = 'Noémie'
```

On distingue trois éléments :

- une valeur, ici la chaîne de caractères `'Noémie'`
- le nom de la variable, ici `prenom`
- un signe = qui sépare le nom de la variable - en partie gauche - de la valeur - en partie droite -

La réalisation de l'affectation se déroule en deux étapes :

1. la valeur de la partie droite est calculée
2. cette valeur devient la (nouvelle) valeur associée à la variable

Suite à l'affectation précédente, nous pouvons observer cette association :

```
>>> prenom
'Noémie'
>>> prenom + ' programme en Python'
'Noémie programme en Python'
>>> "Tu t'appelles " + prenom
"Tu t'appelles Noémie"
```

Il est possible de procéder à une nouvelle affectation d'une variable déjà définie. L'association avec l'ancienne valeur est alors perdue.
Par exemple

```
>>> prenom
'Noémie'
>>> prenom = 'Ernest'
>>> prenom
'Ernest'
```

La valeur utilisée dans l'affectation peut être le résultat d'une opération (plus généralement d'une expression - nous traiterons des expressions sous peu -) :

```
>>> elle = 'Noémie '
>>> lui = 'Ernest '
>>> prenoms = elle + lui
```

```
>>> prenoms
'Noémie Ernest '
```

Cette valeur partie droite peut même utiliser l'ancienne valeur de la variable qui sera affectée :

```
>>> prenoms = ''
>>> prenoms
''
>>> prenoms = prenoms + 'Ernest '
>>> prenoms
'Ernest '
>>> prenoms = prenoms + 'Noémie '
>>> prenoms
'Ernest Noémie '
```

(Remarquons que le signe = utilisé pour l'affectation n'a rien à voir avec le = mathématique ! Que pourrait signifier $prenoms = prenoms +' Noémie'$ en maths ?)

Identificateur

Les **identificateurs** sont les noms choisis par le programmeur, par exemple pour nommer ses variables. Ils doivent respecter des règles définies par le langage Python. Ils doivent aussi être choisis avec soin.

En Python,

1. un identificateur ne peut contenir que des lettres, des chiffres ou le *blanc souligné* `_`
2. un identificateur ne peut pas commencer par un chiffre (et doit donc commencer par une lettre ou par `_`)
3. un identificateur ne peut pas être un des *mots-clés* du langage Python. Ces mots-clés réservés sont les suivants :

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

On évitera les caractères spéciaux et lettres accentuées.

Des identificateurs *significatifs* seront choisis.
En particulier, l'objectif est que ces identificateurs soient immédiatement compréhensibles par le lecteur. On préférera donc par exemple `prenoms` à `p` ou `prn`.

Les identificateurs sont sensibles à la casse, ainsi `ide` et `Ide` sont deux identificateurs différents.

On adoptera un nommage cohérent tout au long du semetre, par exemple de

- nommer les variables avec des lettres minuscules et le blanc souligné, par exemple `les_prenoms`
- nommer les « constantes » avec des lettres majuscules et le blanc souligné, par exemple `NOMBRE_MAX` (une constante est une variable dont la valeur ne sera pas modifiée)

D'autres conventions ou règles d'usages seront introduites plus tard dans le cours.

Mémoire

Lorsque qu'un programme Python est exécuté, les identificateurs et les valeurs associées sont sauvegardées dans un espace mémoire. Ces associations évoluent lors de l'exécution du programme, on parle de l'*état de la mémoire* à un instant donné. La valeur d'une variable, c'est-à-dire son état, est consultable en tapant l'identificateur dans le shell.
Par exemple :

```
>>> prenom = 'Noémie'
>>> prenom
'Noémie'
```

Memento

- une variable est un nom qui désigne une valeur
- l'affectation associe une valeur à une variable
- l'état de la mémoire représente l'ensemble des associations identificateur/valeur d'un programme

Nombres entiers

Coup d'œil

On découvre

- le type `int` utilisé pour représenter les nombres entiers

Le type `int`

Le type *entier* ou `int` de Python permet de représenter n'importe quel nombre relatif.

Les littéraux entiers s'écrivent naturellement avec les dix chiffres `0`, `1`, …, `9` (et donc en base 10).
Par exemple le nombre 42 s'écrira avec la suite de caractères `42`.

```
>>> 42
42
```

Le caractère `-` est utilisé comme préfixe pour les nombres négatifs. Par exemple :

```
>>> -14
-14
```

Il y a cependant quelques règles à respecter :

1. Il ne faut pas utiliser de séparateurs de milliers (ni espace, ni point) : le nombre 10000000 s'écrit en Python `10000000` et non ~~`10 000 000`~~ ou ~~`10.000.000`~~.
2. Il ne faut pas précéder le premier chiffre d'un 0, sauf pour le nombre 0. On doit écrire `7` et non ~~`007`~~.
3. D'autres règles existent. Elles ne sont pas indispensables pour pouvoir écrire en Python n'importe quel nombre entier.

D'autres écritures (en binaire, octal ou hexadécimal) existent.

Opérations

Les opérations arithmétiques habituelles sont disponibles sur les valeurs du type `int` :

- l'addition `a + b`,
- la soustraction `a - b`,
- la multiplication `a * b`, et
- l'élévation à la puissance `a ** b`.

Ces opérations produisent une valeur du type `int`.

```
>>> 3 + 14
17
>>> 3 - 14
-11
>>> 3 *  14
42
>>> 2**3
8
```

Il est possible de mettre aucune, une ou plusieurs espaces entre un opérande et l'opérateur. Utilisez ce qui vous semble le plus lisible.

Notons que l'opérateur de multiplication est le $*$, et non le $\times$, le point, ou la simple juxtaposition. On ne peut écrire `3 × x`, ni `3.x`, ni `3x`.

La division euclidienne est également disponible.
Prenons par exemple la division euclidienne de 51 par 10. On note $51 = 10 \times 5 + 1$ où 5 est appelé le quotient et 1 le reste.
Ce reste est également nommé modulo.
Le schéma ci-dessous illustre cette division posée comme cela fait pour la résoudre à la main.

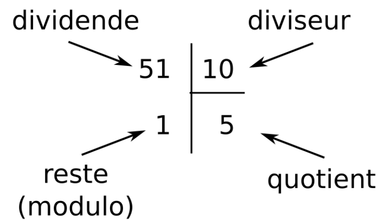


Figure 1: Exemple d'une division euclidienne posée

- le quotient est obtenu par `a // b` (notez bien l'opérateur `//` avec deux barres obliques)
- le reste ou modulo est obtenu par `a % b`

```
>>> 51 // 10
5
>>> type(51 // 10)
<class 'int'>
>>> 51 % 10
1
>>> type(51 % 10)
<class 'int'>
```

Memento

- les valeurs du type `int` représentent les nombres entiers
- on écrit les entiers (en base dix) en respectant quelques règles simples
- les opérations arithmétiques d'addition `+`, soustraction `-`, multiplication `*` et puissance `**` sont disponibles sur les entiers
- le quotient et le reste de la division euclidienne s'obtiennent avec les opérateurs `//` et `%`

Nombres à virgule flottante

Coup d'œil

On découvre

- les *flottants* et le type `float` utilisé pour représenter les nombres réels
- les opérations sur les *flottants*
- comment combiner nombres entiers et nombres flottants

Le type `float`

Le type *flottants* ou `float` de Python permet de représenter certains nombres réels. On dit aussi *nombre flottant* ou *nombre à virgule flottante*.

Nous pouvons considérer qu'un nombre flottant est la représentation d'une approximation d'un nombre réel, de la même façon que le décimal $3,1416$ est une approximation du réel π.

Notez bien que tous les nombres réels ne peuvent pas être représentés par une valeur de type `float`.

Littéraux flottants

Deux notations sont possibles pour les littéraux flottants :

- la notation simple,
- la notation scientifique.

Notation simple

Avec la notation simple, on écrit la partie entière et la partie décimale en les séparant par un point `.`. Le point est le séparateur décimal habituellement utilisé dans les systèmes anglo-saxons, alors que presque tous les autres systèmes utilisent la virgule.

Comme pour les nombres entiers, on peut préfixer un nombre à virgule flottante par un signe moins `-`.

```
>>> 3.1416
3.1416
>>> -1.25
-1.25
```

L'usage du séparateur décimal `.` est obligatoire afin de distinguer les flottants des entiers.

```
>>> 51.
51.0
>>> 51
51
>>> type(51.)
<class 'float'>
>>> type(51)
<class 'int'>
```

Il est possible d'omettre le zéro de la partie entière :

```
>>> .5
0.5
```

Notation scientifique

La notation scientifique permet de représenter facilement des grands ou petits nombres.

Prenons l'exemple du *nombre d'Avogadro*, approximé à $6,02214076 \times 10^{23}$ mol^{-1}.
On appelle *mantisse* la partie à gauche du symbole $\times$, ici $6,02214076$. Il s'agit d'un nombre réel.
On appelle *exposant* la puissance de 10, ici 23. Il s'agit d'un nombre relatif.

Pour la notation scientifique, on écrit successivement :

- la mantisse en notation simple, ici `6.02214076`
- le caractère `e`
- l'exposant, ici `23`. On peut aussi écrire `+23`

Une notation simple du nombre d'Avogadro est `602214076000000000000000.0`, longue et peu lisible. On écrira donc plutôt :

```
>>> 6.02214076e+23
6.02214076e+23
>>> 6.02214076e23
6.02214076e+23
>>> 602214076000000000000000.0
6.02214076e+23
```

Opérations flottantes

Les opérations arithmétiques habituelles sont disponibles sur les valeurs du type `float` :

- l'addition `x + y`, la soustraction `x - y`, la multiplication `x * y`, et l'élévation à la puissance `x ** y`.

Ces opérations produisent une valeur du type `float`.

```
>>> 3.0 + 0.14
3.14
>>> 3.0 - 1.4
1.6
>>> 3.0 * 14.
42.0
>>> 9.0 ** 0.5
3.0
```

La division flottante est également disponible, notée de la simple barre oblique : `x / y`.
Elle produit une valeur flottante.

```
>>> 5.0 / 2.0
2.5
>>> 4.0 / 2.0
2.0
>>> 1.1 / 0.1
11.0
```

Approximation des nombres réels

Les flottants sont des approximations, certains résultats peuvent paraître surprenants.

```
>>> 0.1 + 0.2
0.30000000000000004
```

Le résultat du calcul précédent est précis à 10^{-17} près. Pour en savoir plus, vous pouvez consulter la documentation Python

Opérations entre entiers et flottants

L'ensemble des opérations arithmétiques s'appliquent à des nombres entiers et flottants.

Pour toutes les opérations sauf la division flottante, le type du résultat :

- est `int` si les deux opérandes sont des `int`
- est `float` si les deux opérandes sont des `float`
- est `float` si une opérande est un `int` et l'autre un `float`

```
>>> 3 * 14
42
>>> 3.0 * 14.0
42.0
>>> 3.0 * 14
42.0
>>> 3 * 14.0
42.0
```

La division flottante / produit toujours une valeur de type `float`, même si les deux opérandes sont de type `int` :

```
>>> 5 // 2
2
>>> 5 / 2
2.5
>>> 4 // 2
2
>>> 4 / 2
2.0
```

Memento

- les nombres flottants sont des représentations ou approximations des nombres réels
- le type `float` est utilisé pour représenter les nombres flottants
- la notation simple des flottants de la forme `123.45` utilise le point comme séparateur décimal
- la notation scientifique des flottants de la forme `12.34e+56` distingue mantisse et exposant
- les opérations arithmétiques d'addition `+`, soustraction `-`, produit `*`, division euclidienne `//` et `%`, et puissance `**` sont disponibles sur les flottants et les entiers
- la division flottante notée `/` produit toujours une valeur flottante

Expressions et instructions

Coup d'œil

On découvre

- comment combiner plusieurs opérations pour former des expressions
- d'autres opérations diverses

On précise

- la notion d'instruction et d'état

Expressions

Il est possible de combiner plusieurs opérateurs pour former une *expression*.
On écrit par exemple `3 + 4 - 2`.

En programmation, une expression est la notation d'un calcul à réaliser.
Lors de l'exécution, l'expression sera toujours réduite à une valeur.
On dit qu'elle est *évaluée*.

Cette évaluation se fait en suivant un ordre de priorité qui définit l'ordre des opérations à effectuer.

Par exemple, l'évaluation de `3 + 4 - 2`

- calculera la somme de `3 + 4`
- puis fera la différence entre cette valeur et `2`

Le tableau ci dessous classe les opérateurs par ordre de priorité décroissant. Les opérateurs sur la même ligne ont la même priorité.

	**		
*	/	//	%
	+	-	

```
>>> 2 * 3 + 5 * 1
11
>>> 2 ** 2 * 2
8
```

Lorsque deux opérateurs ont la même priorité, ils sont évalués de gauche à droite sauf les `**` qui sont évalués de droite à gauche.

```
>>> 2 - 3 - 1
-2
>>>  5 // 2 * 2
4
>>> 2 ** 2 ** 3
256
```

Il est possible de contrôler l'ordre d'évaluation des opérations en utilisant des parenthèses `(` et `)` :

```
>>> 2 * (3 + 5) * 1
16
```

```
>>> 2 - (3 - 1)
0
>>> (2 ** 2) ** 3
64
```

On ne se privera pas d'utiliser les parenthèses.

Instructions

Une instruction exprime un ordre. Son exécution va modifier l'état de la machine, de l'ordinateur.

L'état est essentiellement l'ensemble des valeurs associées aux variables définies.

La seule instruction que nous connaissons pour le moment est l'affectation. Elle modifie la valeur associée à une variable, donc l'état.

(La position du curseur à l'écran et d'autres choses font aussi parties de l'état.)

Remarquons qu'une expression n'est pas une instruction. Elle ne modifie en effet pas l'état.

Un programme est une suite d'instructions.
L'exécution du programme consiste en l'exécution successive des instructions pour faire passer l'état de l'état initial à un état final.

Autres opérations

Répétition de chaînes de caractères

L'opérateur $*$ appliqué à une valeur entière n et une chaîne de caractères s :

- produit une chaîne de caractères
- résultat de n concaténations de la chaîne s.

```
>>> triple = 3
>>> triple * 'bravo ! '
'bravo ! bravo ! bravo ! '
>>> 'bravo !' * 3
'bravo !bravo !bravo !'
```

D'autres opérations sur les chaînes de caractères seront présentées plus tard.

Remarquons que les expressions peuvent être formées d'opérateurs de toute sorte :

```
>>> double = 2
>>> double * ('bravo,' + ' ') + 'bravo !'
'bravo, bravo, bravo !'
```

Conversions

Des fonctions prédéfinies du langage Python permettent de changer le type d'une valeur, lorsque c'est possible. Ce changement s'appelle une conversion de type.

La fonction prédéfinie `str()` produit une représentation sous forme d'une chaîne de caractères de la valeur passée en paramètre, par exemple d'une valeur numérique.

```
>>> str(12)
'12'
>>> str(42.14)
'42.14'
```

Réciproquement, la fonction prédéfinie `int()` produit un nombre entier à partir d'une chaîne de caractères, si et seulement si cette chaîne ne contient que des chiffres, éventuellement précédés du signe `-`. Si un autre caractère est présent, à part les espaces qui sont tolérés, une erreur est affichée.

```
>>> int('12')
12
>>> int('007')
```

```
7
>>> int('-12 ')
-12
>>>  int('12.')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.'
>>> int('12.')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.'
>>> int('23.5')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
>>> int('24°C')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '24°C'
```

Enfin, la fonction prédéfinie `float()` produit un nombre flottant à partir d'une chaîne de caractères, si et seulement si cette chaîne ne contient que des chiffres, éventuellement un `.` et éventuellement précédés du signe `-`. Si un autre caractère est présent, à part les espaces qui sont tolérés, une erreur est affichée.

```
>>> float('12')
12.0
>>> float('007')
7.0
>>> float('-12 ')
-12.0
>>> float('12.')
12.0
>>> float('23.5')
23.5
>>> float('24°C')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: could not convert string to float: '24°C'
```

Memento

- une expression combine une ou plusieurs opérations
- l'évaluation d'une expression la réduit à une valeur
- des règles de priorités fixent l'ordre d'application des opérations
- les parenthèses permettent de préciser l'ordre d'application des opérations
- une instruction modife l'état de l'ordinateur
- l'affectation est une instruction qui modifie l'association des valeurs aux variables
- des fonctions pré-définie permettent de convertir une valeur d'un type en un autre type, sous certaines conditions.

Exercices niveau débutant

Expressions arithmétiques

Pour les questions qui suivent, n'utilisez pas tout de suite l'interpréteur Python (ni une calculatrice).

- Calculez les valeurs des expressions et précisez leur type.
- Vérifiez, ensuite la validité de votre réponse en utilisant l'interpréteur et la fonction `type`

1. Quelques expressions.

```
10 + 2 * 7
(10 + 2) * 7
35. - 5 * 8
-5-5
+5-5
5+-5
5--5
```

2. Indiquez la signification des trois opérateurs /, // et %.

3. Quelle est la différence entre les expressions ci-dessous ? Ont-elles la même valeur ? le même type ?

```
5 / 2
5 // 2
5 % 2
5 / 2.
5 // 2.
```

Variables et affectation

4. Sans utiliser l'interpréteur Python, prévoyez si ces noms de variable sont valides ou non.

`n`	`mon_ue`	`dame`
`test`	`mon ue`	`roi`
`$test`	`mon-ue`	`as`
`test1`	`monUe`	`avec`
`1test`	`monUE`	`with`
`_`	`Mon_Ue`	`sans`
		`without`

5. Décrivez l'évolution des valeurs associées aux variables au fur et à mesure de l'exécution des instructions suivantes.
   Donnez les valeurs finales de chaque variable.

   En d'autres termes, décrivez l'état de la mémoire après chaque instruction.

```
a = 10
b = a
a = 12
b = b + 1
```

```
x = 1
y = 2
z = y
y = x
x = z
```

6. Affectez aux identificateurs `a`, `b`, `c` et `d` respectivement les valeurs 1, 2, 3 et 4 puis transcrivez, en expressions arithmétiques Python3, les formules mathématiques suivantes.

   - $(a+b)^2$, vous devez trouver 9
   - $a^2+2ab+b^2$, vous devez trouver 9
   - $\frac{a+b}{c+d}$, vous devez trouver environ $0,42857$
   - $\frac{1}{a+1}$, vous devez trouver $0,5$

7. Consultez les modalités de calcul de la note finale de l'UE *Informatique*.
   Écrivez les instructions nécessaires à son calcul à l'aide de variables et observez la note finale obtenue avec différents jeux de notes intermédiaires.

Chaînes de caractères

8. Affectez à des variables les chaînes de caractères suivantes :

   - C'est l'automne !
   - Il fait 18°C.
   - Ceci " est un guillemet.
   - Ce caractère " n'est-il pas un guillemet ?

9. Calculez les valeurs des expressions suivantes et précisez leur type. Vérifiez ensuite la validité de votre réponse en utilisant l'interpréteur et la fonction `type`.

   Comme dans un exercice précédent, n'utilisez pas tout de suite l'interpréteur Python.

```
'5' + '5'
5 + '5'
'5' - '5'
5 * '5'
'5' * '5'
```

Exercices niveau intermédiaire

Échange de valeurs

10. Soient les deux variables définies suite à l'exécution des instructions suivantes

```
n1 = 'Un'
n2 = 'Deux'
```

Quelles sont les nouvelles valeurs associées à ces variables suite à l'exécution de

```
n2 = n1
n1 = n2
```

Proposez une méthode pour échanger les valeurs des deux variables : à la fin, `n1` doit contenir la valeur *'Deux'* et `n2` doit contenir *'Un'*.

Cette méthode doit bien entendu fonctionner quelles que soient les valeurs des variables.

11. Écrivez maintenant une suite d'instructions pour effectuer une permutation circulaire avec 3, puis 4 variables : la première variable prend la valeur de la dernière, la dernière de la précédente, ainsi de suite. Par exemple pour les trois variables

```
n1 = 'Un'
n2 = 'Deux'
n3 = 'Trois'
```

à l'issue de la permutation, `n1` sera associée à la valeur *'Trois'*, `n2` à la valeur *'Un'*, et `n3` à la valeur *'Deux'*.

Programme de calcul

12. Donner une suite d'instructions pour effectuer les calculs suivants.
    - soit un nombre
    - lui ajouter 3
    - multiplier par 2
    - ajouter le nombre initial
    - soustraire 12
    - diviser par 3
    - ajouter 2

Calculs d'intérêts

Supposez que vous disposez d'un capital d'un million d'euros (on peut rêver), et que vous l'avez investi dans un placement qui vous rapporte 5% d'intérêt par an. Supposez enfin que ces intérêts sont ajoutés au capital tous les six mois. De quel capital disposerez-vous dans cinq ans ?

La formule pour calculer le montant final du capital est la suivante :

$$A = C(1 + \frac{r}{n})^{nt}$$

où

- C est le capital initial
- r est le taux d'intérêt
- t est le nombre d'années
- n est le nombre de fois que les intérêts sont calculés et ajoutés au capital chaque année (par exemple, $n = 2$ dans la situation décrite ci-dessus).

13. Définissez les variables `C`, `r`, `t` et `n`, puis calculez le montant final dans une variable `A_t5_n2` selon la formule ci-dessus.

    Vous devez trouver environ $1280084, 54$

14. Refaites le calcul en supposant que les intérêts sont capitalisés tous les trimestres et affectez la valeur obtenue à une variable `A_t5_n4`. Quelle est la différence entre les deux capitaux résultants ?

    Vous devez trouver environ $1282037, 23$

Chiffres et divisions euclidiennes

15. Donnez une expression calculant le chiffre des unités de 2031. (Vous devez trouver `1`.)
16. Donnez une expression calculant le nombre de dizaines de 2031. (Vous devez trouver `203`.)
17. Donnez une expression calculant le chiffre des dizaines de 2031. (Vous devez trouver `3`.)
18. Définissez une variable `nip` contenant votre NIP - numéro étudiant - (avec le type entier), une variable `n` contenant l'entier 5, puis écrivez une expression utilisant ces deux variables et calculant le n-ième chiffre en partant de la droite de votre NIP.

Dans l'ordre

19. Soient les instructions Python suivantes :

```
str1 = 'oh '
str2 = str2 + '!'
str2 = coeff * str1
coeff = 3
```

    Mettez ces instructions dans l'ordre pour que suite à leur exécution, la valeur associée à la variable `str2` soit « `'oh oh oh !'` ».

20. Même chose avec l'ensemble suivant d'instructions :

```
str1 = 'oh '
str1 = 'oh !'
str2 = str2 + '!'
coeff3 = coeff2 - 1
str2 = coeff2 * str1
coeff2 = 3
```

    Vous devez utiliser toutes les instructions.

2. Fonctions

Utilisation de fonctions

Coup d'œil

On découvre

- la notion de fonction
- comment réaliser un appel de fonction et récupérer la valeur de retour
- quelques fonctions prédéfinies de Python
- comment accéder à la documentation d'une fonction

Fonction (en informatique)

Fonction ≡ Algorithme

données ⟶ [algorithme] ⟶ données

Fonction

- prend des données en entrée – ses **paramètres**, appelés aussi arguments
  - ce sont des valeurs
- produit une valeur – son **résultat**
  - on dit aussi *renvoie* un résultat

Spécification ≡ contrat avec l'utilisateur de la fonction.

Spécification

- description de l'objectif, de ce que fait la fonction
- description des attendus sur les paramètres (types, contraintes particulières)
- description de la valeur renvoyée (type, particularités)

Une fonction est un sous-programme qui permet d'organiser son code. Elle est une suite d'instructions qui sert à faire des actions précises. Le but des fonctions est de simplifier un programme pour le rendre plus lisible, pour ne pas avoir à retaper le même code plusieurs fois dans un même programme, pour éviter les erreurs.

Appel de fonction

Python, comme de nombreux langages, propose des fonctions prédéfinies. Elles sont utilisables directement.

Par exemple, la fonction `len()` renvoie le nombre de caractères d'une chaîne de caractères.

On utilise une fonction via un *appel* : on écrit le nom de la fonction suivi des paramètres entre parenthèses.

Par exemple pour cette fonction `len()`

```
>>> len('informatique')
12
```

On obtient bien le nombre de caractères de la chaîne `'informatique'`

L'appel de la fonction est réalisé ainsi :

1. les paramètres sont évalués, c'est-à-dire que les expressions sont réduites à une valeur.

2. le code de la fonction est exécuté.
3. la valeur de l'appel de fonction est la valeur renvoyée par la fonction.

Sur cet autre exemple :

```
>>> len('info' + 'r' + 'matique')
12
```

1. l'expression qui correspond au paramètre - `'info' + 'r' + 'matique'` - est évaluée. On obtient donc la valeur `'informatique'`.
2. le code qui correspond à la fonction `len()` est exécuté avec comme paramètre cette valeur `'informatique'`.
3. cet appel de fonction produit la valeur `12`

Bien entendu, cette valeur renvoyée par la fonction peut être affectée à une variable, ou utilisée dans une autre expression.

Par exemple

```
>>> nbcar = len('informatique')
>>> nbcar
12
```

ou

```
>>> d = len('informatique') - len('Python')
>>> d
6
```

Une fonction peut aussi accepter plusieurs paramètres, voir un nombre variable de paramètres. Lors d'un appel à la fonction, ils seront séparés par des virgules.

Par exemple, la fonction prédéfinie `min()` renvoie le plus petit de ses paramètres. On peut donc écrire

```
>>> min(12, 6)
6
>>> min(3, 4, 1, 2, 4)
1
```

Une fonction peut n'accepter aucun paramètre. On écrit alors simplement le nom de fonction suivi de parenthèses.

Par exemple, la fonction `dir()` renvoie une liste des variables (pré-)définies :

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'd'
```

Une autre fonction prédéfinie, `str()` que nous avons déjà rencontrée, produit une représentation sous forme d'une chaîne de caractères de la valeur passée en paramètre, par exemple d'une valeur numérique. On a

```
>>> str(12)
'12'
>>> str(42.14)
'42.14'
```

En combinant plusieurs appels de fonction, on peut écrire :

```
>>> 1 + len(str(2 * min(12, 6)))
3
```

→ Donner les différentes étapes de l'exécution de cette instruction.

Documentation

Les fonctions sont généralement documentées : les programmeurs décrivent brièvement la spécification de la fonction.

Il est possible d'accéder à la documentation à l'aide de la fonction `help()`. Pour obtenir l'aide sur la fonction prédéfinie `abs()`, on écrira

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Cette aide est interactive si elle tient sur plusieurs pages :

- on passe à la page suivante avec la touche « *espace* »
- on quitte l'aide en utilisant la touche « `q` »

Memento

- une fonction met en œuvre un algorithme dont on connaît la spécification
- un appel de fonction correspond à une exécution de la fonction
- on appelle une fonction en fournissant son nom suivi d'une série de valeurs entre parenthèses
- ces valeurs sont les paramètres de l'appel
- une fonction accepte un certain nombre de paramètres, éventuellement aucun
- un appel de fonction renvoie une valeur, dite valeur de retour
- Python propose des fonctions prédéfinies
- les fonctions sont documentées. La fonction prédéfinie `help()` permet de consulter cette documentation

Bibliothèque et importation de fonctions

Coup d'œil

On découvre

- la notion de module ou bibliothèque de fonctions
- comment importer et utiliser les fonctions d'un module
- quelques fonctions de modules couramment utilisés

Modules Python

Les fonctions prédéfinies sont accessibles directement en Python. D'autres fonctions sont proposées via des **bibliothèques**. On parle aussi de **modules**.

Python fournit de très nombreux modules. Nous en découvrirons quelques-uns au fur et à mesure de ce semestre.

Voyons ci-dessous un bref aperçu de deux modules que nous utiliserons dans nos exemples.

Le module `math`

Le module `math` de Python contient les définitions de nombreuses fonctions mathématiques telles que *sinus* – `sin()` –, *cosinus* – `cos()` -, *tangente* – `tan()` –, *racine carrée* – `sqrt()` –, etc.
Il contient également des constantes telles que π, noté `pi`.

Le module `random`

Le module `random` de Python regroupe les fonctions de hasard (tirage au sort, génération aléatoire). Nous utiliserons les fonctions

`randint()` qui accepte deux paramètres de type entier - bornes inférieure et supérieure d'un intervalle - et renvoie un entier, choisi au hasard dans cet intervalle, bornes comprises.

`choice()` qui accepte par exemple une chaîne de caractères et renvoie une des lettres de la chaîne, choisie au hasard.

D'un appel à l'autre ces fonctions renvoient donc possiblement une valeur différente.

Importer les fonctions d'un module

L'utilisation d'une fonction d'un module nécessite une importation préalable.

Il est possible

- d'importer un module
- d'importer une fonction particulière d'un module
- d'importer toutes les fonctions d'un module

L'importation n'est réalisée qu'une fois, habituellement en début de programme ou de session. De multiples appels à la fonction sont ensuite possibles.

Importer un module

Pour importer une module, par exemple le module `random`, on utilise la syntaxe suivante

```
import random
```

On identifiera les fonctions importées en préfixant leur nom par celui du module. Par exemple

```
>>> random.choice('aeiouy')
'u'
```

ou

```
>>> random.randint(12, 42)
33
```

Importer une fonction particulière d'un module

Pour importer une fonction d'un module, par exemple la fonction `choice()` du module `random`, on utilise la syntaxe suivante

```
from random import choice
```

On identifiera simplement la fonction importée par son nom. Par exemple

```
>>> choice('aeiouy')
'y'
```

Remarquons que les autres fonctions du module ne sont pas disponibles :

```
>>> randint(12, 42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'randint' is not defined
```

Le message d'erreur indique que le nom `randint` n'est pas défini.

Importer toutes les fonctions d'un module

On peut vouloir importer l'ensemble des fonctions d'un module. On utilise alors la syntaxe

```
from random import *
```

On identifiera simplement les fonctions importées par leur nom. Par exemple

```
>>> choice('aeiouy')
'o'
```

et

```
>>> randint(12, 42)
17
```

→ On pourra vérifier que d'autres fonctions du module `random` ont aussi été importées. Par exemple via un appel à `dir()` ou dans le volet « Variables » de notre environnement de programmation Thonny.

On évitera d'utiliser ce moyen d'importation qui « pollue » l'espace de noms : de nombreuses fonctions deviennent disponibles alors que nous n'allons pas les utiliser.

Memento

- un module ou bibliothèque regroupe un ensemble de fonctions
- il faut importer les fonctions d'un module avant de pouvoir les utiliser
- il existe plusieurs syntaxes pour importer une ou des fonctions
- une fois importée, l'appel à une fonction d'un module se fait sous la forme `fonction()`
- l'appel à une fonction d'un module se fait parfois sous la forme `module.fonction()`

Définition de fonction

Coup d'œil

On découvre
- comment créer un bloc d'instructions
- comment définir nos propres fonctions
- pourquoi définir des fonctions
- comment décrire la spécification et la documentation de nos fonctions

Indenter un bloc de code

Une série d'instructions peut être regroupée en un bloc. Dans de nombreux langages, des mots-clés – `begin` et `end` par exemple –, ou des accolades `{` et `}` marquent ces blocs. En Python, on identifie les instructions faisant partie d'un même bloc par le fait qu'elles sont écrites décalées vers la droite à un même niveau. On dit qu'elles sont *indentées*.

Il est possible d'utiliser le nombre d'espaces que l'on veut pour indenter un bloc. Par contre, ce nombre doit toujours être le même au sein d'un programme. Il est cependant conseillé d'utiliser 4 espaces, 8 pour les blocs imbriqués au second niveau, etc.

On n'utilisera pas la tabulation pour l'indentation.

La fin du bloc est marquée par la fin de l'indentation, plus précisément par la reprise de l'indentation précédente.

Sur un exemple

```
instruction bloc niveau 0
instruction bloc niveau 0
    instruction bloc niveau 1
    instruction bloc niveau 1
        instruction bloc niveau 2
        instruction bloc niveau 2
        instruction bloc niveau 2
    instruction bloc niveau 1
instruction bloc niveau 0
```

Notre première fonction

Au delà des fonctions prédéfinies et des fonctions des bibliothèques, il est possible de définir nos propres fonctions. Voyons comment procéder sur un exemple.

La définition d'une fonction est introduite par le mot-clé `def` suivi du nom de la fonction et d'une liste d'identificateurs. Ce sont les paramètres.

Cette première ligne se termine par un `:`. On écrit donc par exemple :

```
def celsius2fahrenheit(cels):
```

Les lignes suivantes sont le corps de la fonction et constituent un bloc d'instructions. Elles seront indentées d'un niveau.

La première instruction de la fonction peut être une **docstring** – *documentation string* –, la chaîne de caractères documentation de la fonction. Elle est délimitée par de triples guillemets.

Notre fonction débute donc par :

```
def celsius2fahrenheit(cels):
    """Convertit en Fahrenheit une température en degrés Celsius

    Paramètre :
    - cels : float, température en degrés Celsius
    Valeur de retour :
    - float (température en degrés Fahrenheit)
    """
```

Le corps de la fonction lui-même, c'est-à-dire les instructions qui suivent, seront exécutées à chaque appel de la fonction.

Dans ces instructions, les identificateurs des paramètres sont utilisés pour accéder aux valeurs fournies lors de l'appel.

On écrira par exemple

```
def celsius2fahrenheit(cels) :
    """Convertit en Fahrenheit une température en degrés Celsius

    Paramètre :
    - cels : float, température en degrés Celsius
    Valeur de retour :
    - float (température en degrés Fahrenheit)
    """
    fahr = ((9 * cels) / 5) + 32
    return fahr
```

L'instruction `return` met fin à l'exécution de la fonction, quelque-soit sa position dans le bloc d'instruction de la fonction. La valeur fournie à `return` est la valeur qui sera renvoyée par la fonction.

Suite à notre définition de fonction, nous pouvons

- consulter la documentation de la fonction :

  ```
  >>> help(celsius2fahrenheit)
  Help on function celsius2fahrenheit in module __main__:

  celsius2fahrenheit(cels)
      Convertit en Fahrenheit une température en degrés Celsius

      Paramètre :
      - cels : float, température en degrés Celsius
      Valeur de retour :
      - float (température en degrés Fahrenheit)
  ```

- faire des appels à cette fonction :

  ```
  >>> celsius2fahrenheit(24)
  75.2
  ```

Composer des fonctions

Par composition de fonctions on élabore des algorithmes de plus en plus complexes.

Cette démarche repose sur de grands principes élémentaires de programmation

- décomposition
- réutilisation
- modularité
- encapsulation

Définissons une deuxième fonction :
En montagne, plus on s'élève, plus il fait froid. La température baisse de 0,65°C tous les 100 mètres.

```
def delta(alt) :
    """Température perdue à une altitude donnée en mètres"""
    return 0.65 * alt / 100
```

Et une troisième qui utilise les deux précédentes :

```
def fahr_alt(t_ref, alt) :
    """Température en Fahrenheit à une altitude donnée, pour une température
    de référence en Celsius

    Paramètres :
    - t_ref : température de référence en Celsius
    - alt : altitude en mètres
    Valeur de retour :
    - float (température en Fahrenheit à l'altitude donnée)
    """
    cels = t_ref - delta(alt)
    fahr = celsius2fahrenheit(cels)
    return fahr
```

On peut alors calculer la température en Fahrenheit à 3000 m d'altitude étant donné qu'il fait 24°C ici :

```
>>> temp_ici = 24
>>> fahr_alt(temp_ici, 3000)
40.1
```

Documenter nos fonctions

La documentation de nos fonctions est fournie sous la forme d'une simple chaîne de caractères.

On précisera particulièrement le sens des paramètres, le type attendu pour ces paramètres, et les contraintes d'utilisation, le type de la valeur renvoyée.

La documentation des fonctions définies dans les exemples précédents comporte certains de ces éléments. Nous pouvons la compléter, en particulier en ce qui concerne les contraintes d'utilisation :

Le calcul de la variation de température en fonction de l'altitude tel que décrit précédemment n'est plus valable quand on atteint la stratosphère, disons 12 km. Par ailleurs, une température ne peut être inférieure à -273,15°C.

```
def celsius2fahrenheit(cels) :
    """Convertit en Fahrenheit une température en degrés Celsius

    Paramètre :
    - cels : float, température en degrés Celsius
    Valeur de retour :
    - float (température en degrés Fahrenheit)
    Contrainte :
    - la température cels doit être strictement supérieure à -273,15°C
    """
    fahr = ((9 * cels) / 5) + 32
    return fahr

def delta(alt) :
    """Température perdue à une altitude donnée en mètres

    Paramètre :
    - alt : altitude en mètres
    Valeur de retour :
    - float (variation de température due à l'altitude)
    Contrainte :
    - l'altitude alt doit être inférieure à 12 000 m
```

```python
    """
    return 0.65 * alt / 100
```

L'utilisateur des fonctions doit se plier à la documentation, et ne pas faire d'appel à la fonction avec des valeurs de paramètres qui ne respecteraient pas les contraintes.

Memento

- on utilise l'indentation pour identifier les blocs d'instructions
- le mot-clé `def` introduit la définition d'une fonction
- les paramètres sont des identificateurs qui permettent d'accéder aux valeurs passées à la fonction
- un bloc d'instructions forme le corps de la fonction
- l'instruction `return` met fin à l'exécution de la fonction
- la valeur renvoyée par la fonction est celle associée au `return`
- on compose des fonctions pour en former de plus complexes
- une documentation sous forme de docstring décrit la spécification de la fonction
- cette documentation précise le type des paramètres et de la valeur de retour, et éventuellement des contraintes sur les valeurs attendues

Test de fonction

Coup d'œil

On découvre

- comment compléter la spécification et la documentation des fonctions par des exemples et tests
- comment automatiser les tests de nos fonctions

Une fonction `plus_grande_racine()`

Nous désirons définir une fonction qui renvoie la plus grande racine d'un polynôme du second degré $ax^2 + bx + c$.

La fonction acceptera trois paramètres : les valeurs des coefficients a, b, et c.

Cette plus grande racine est évidemment égale à :

$$max(\frac{-b+\sqrt{\Delta}}{2a}, \frac{-b-\sqrt{\Delta}}{2a})$$

avec

$$\Delta = b^2 - 4ac$$

pourvu que $\Delta \geq 0$.

Prototype et documentation

La première étape est de définir l'interface et la documentation de notre fonction `plus_grande_racine()`. On écrit par exemple :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    """
    pass
```

L'exécution de l'instruction `pass` ne produit rien. Elle est utilisée pour remplacer du code qui n'est pas encore écrit car elle évite la génération d'une erreur lors de l'exécution d'un programme incomplet.

Contraintes d'utilisation

L'utilisation de cette fonction `plus_grande_racine()` nécessite le respect de deux contraintes, appelées aussi conditions d'utilisation :

- $b^2 - 4ac \geq 0$
- $a \neq 0$

Il nous faut l'indiquer dans la documentation de la fonction. On écrira donc :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    Contraintes :
    - a doit être non nul
    - (b**2 - 4*a*c) doit être positif ou nul
    """
    pass
```

Exemples d'utilisation

La documentation peut aussi être enrichie d'exemples d'utilisation de la fonction.

Si nous avions complètement défini notre fonction, nous pourrions l'utiliser :

```
>>> plus_grande_racine(1, 1, -6)
2.0
>>> plus_grande_racine(5, 0, -5)
1.0
>>> plus_grande_racine(2, 2, 0)
0.0
```

Ces codes Python d'appel de la fonction et de résultats attendus permettent

- à un utilisateur de la fonction, d'éventuellement mieux comprendre la spécification de la fonction,
- à un programmeur de la fonction, de s'assurer que le code de sa fonction produit un résultat correct sur ces quelques exemples.

Tests unitaires

Un jeu de tests est un ensemble de jeux de paramètres et des résultats attendus.

Un **test unitaire** d'une fonction est la vérification que l'exécution de la fonction sur un jeu de tests produit bien les résultats attendus.

Les jeux de paramètres seront choisis avec soin, par exemple pour inclure les cas limites ou des cas particuliers.

Documenter avec des tests

On ajoutera à la documentation de la fonction une section `Exemples` qui reprendra une session d'appels de la fonction et de résultats sur un jeu de tests.

Par exemple :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
```

```
    Contraintes :
    - a doit être non nul
    - (b**2 - 4*a*c) doit être positif ou nul
    Exemples :
    >>> plus_grande_racine(1, 1, -6)
    2.0
    >>> plus_grande_racine(5, 0, -5)
    1.0
    >>> plus_grande_racine(2, 2, 0)
    0.0
    """
    pass
```

Notons que cette section `Exemples` de la documentation doit être écrite *avant* d'écrire la fonction. Il ne s'agit pas d'écrire la fonction, d'exécuter des appels à la fonction, et de recopier le résultat.

Le module `doctest`

Le module `doctest` permet d'effectuer automatiquement les tests inclus dans la documentation `docstring`.

Pour lancer automatiquement les tests, il suffit d'utiliser la fonction *ad hoc* de ce module :

```
>>> import doctest
>>> doctest.testmod()
```

Un premier test — échec

Avec la définition incomplète de notre fonction, nous obtenons bien entendu des erreurs :

```
>>> import doctest
>>> doctest.testmod()
**********************************************************************
File "__main__", line 13, in __main__.plus_grande_racine
Failed example:
    plus_grande_racine(1, 1, -6)
Expected:
    2.0
Got nothing
**********************************************************************
File "__main__", line 15, in __main__.plus_grande_racine
Failed example:
    plus_grande_racine(5, 0, -5)
Expected:
    1.0
Got nothing
**********************************************************************
File "__main__", line 17, in __main__.plus_grande_racine
Failed example:
    plus_grande_racine(2, 2, 0)
Expected:
    0.0
Got nothing
**********************************************************************
1 items had failures:
   3 of   3 in __main__.plus_grande_racine
***Test Failed*** 3 failures.
TestResults(failed=3, attempted=3)
```

Un second test — succès

En complétant notre fonction par un code « correct », nous obtenons :

```
>>> doctest.testmod()
TestResults(failed=0, attempted=3)
```

Un test, en détail

Nous pouvons demander plus de détails à l'aide du paramètre `verbose` :

```
doctest.testmod(verbose=True)
Trying:
    plus_grande_racine(1, 1, -6)
Expecting:
    2.0
ok
Trying:
    plus_grande_racine(5, 0, -5)
Expecting:
    1.0
ok
Trying:
    plus_grande_racine(2, 2, 0)
Expecting:
    0.0
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.plus_grande_racine
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=3)
```

Systématiser les tests

Mieux, il est possible de réaliser les tests à chaque fois que l'on exécute un fichier Python. Il suffit d'ajouter à la fin du fichier les 3 lignes

```
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Tous les `doctest` des fonctions définies dans le fichier seront exécutés systématiquement lors de l'exécution du fichier.

Memento

- la spécification des fonctions est complétée par ses contraintes d'utilisation
- la chaîne de documentation `dosctring` comporte une section `Contraintes`
- un jeu de tests est un ensemble de jeux de paramètres et résultats attendus de l'exécution d'une fonction
- la chaîne de documentation `dosctring` inclut un jeu de tests dans une section `Exemples:`
- le test unitaire est la vérification de la bonne exécution de ces tests
- le module `doctest` permet d'automatiser et systématiser l'exécution des tests unitaires

Test de fonction avec L1test

Coup d'œil

On découvre

- comment compléter la spécification et la documentation des fonctions par des exemples et tests
- comment automatiser les tests de nos fonctions

Une fonction `plus_grande_racine()`

Nous désirons définir une fonction qui renvoie la plus grande racine d'un polynôme du second degré $ax^2 + bx + c$.

La fonction acceptera trois paramètres : les valeurs des coefficients a, b, et c.

Cette plus grande racine est évidemment égale à :

$$max(\frac{-b+\sqrt{\Delta}}{2a}, \frac{-b-\sqrt{\Delta}}{2a})$$

avec

$$\Delta = b^2 - 4ac$$

pourvu que $\Delta \geq 0$.

Prototype et documentation

La première étape est de définir l'interface et la documentation de notre fonction `plus_grande_racine()`. On écrit par exemple :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    """
    pass
```

L'exécution de l'instruction `pass` ne produit rien. Elle est utilisée pour remplacer du code qui n'est pas encore écrit car elle évite la génération d'une erreur lors de l'exécution d'un programme incomplet.

Contraintes d'utilisation

L'utilisation de cette fonction `plus_grande_racine()` nécessite le respect de deux contraintes, appelées aussi conditions d'utilisation :

- $b^2 - 4ac \geq 0$
- $a \neq 0$

Il nous faut l'indiquer dans la documentation de la fonction. On écrira donc :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    Contraintes :
    - a doit être non nul
    - (b**2 - 4*a*c) doit être positif ou nul
    """
    pass
```

Exemples d'utilisation

La documentation peut aussi être enrichie d'exemples d'utilisation de la fonction.

Si nous avions complètement défini notre fonction, nous pourrions l'utiliser dans l'interpréteur et obtenir les résultats suivants:

```
>>> plus_grande_racine(1, 1, -6)
2.0
>>> plus_grande_racine(4, -4, 1)
0.5
>>> plus_grande_racine(2, 2, 0)
0.0
```

Nous reconnaissons ici l'invite `>>>` de l'interpréteur Python telle qu'elle apparaît dans la console de Thonny.

Ces exemples associant appel de la fonction et résultat attendu sont précieux dans la documentation, car ils permettent à un utilisateur de la fonction d'éventuellement mieux comprendre la spécification de la fonction.

On va les écrire un peu différemment de telle sorte que le programmeur de la fonction puisse la **tester** automatiquement : c'est à dire s'assurer de manière automatique que, sur chacun de ces quelques exemples, le code de sa fonction produit bien le résultat attendu en réponse à l'évaluation de l'appel de la fonction.

L'invite de l'interpréteur Python sera remplacée par l'invite de la **bibliothèque de test L1test** utilisée dans l'UE, au choix :

- `$$$`
- `$py`
- `$PY`

On pourra donc obtenir :

```
def plus_grande_racine (a, b, c):
    """Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    Contraintes :
    - a doit être non nul
    - (b**2 - 4*a*c) doit être positif ou nul
    Exemples :
    $$$ plus_grande_racine(1, 1, -6)
```

```
    2.0
    $$$ plus_grande_racine(4, -4, 1)
    0.5
    $$$ plus_grande_racine(2, 2, 0)
    0.0
    """
    pass
```

Tests unitaires, L1 test

Un **L1test**, ou simplement **un test**, est écrit dans la docstring de la fonction testée. Dans sa version la plus simple, il se compose de 2 lignes se suivant immédiatement :

```
$$$ <expression_à_évaluer>
<expression_résultat_attendu>
```

La documentation de la fonction `plus_grande_racine` contient donc 3 tests (on parle d'un **jeu de tests**). Pour le premier test, l'expression à évaluer est `plus_grande_racine(1, 1, -6)` et le résultat attendu est `2.0`. Les valeurs des paramètres (ici `1`, `1` et `-6`) sont appelées des **données de test**.

Éxécuter un test déclenche les évaluations suivantes:

- évaluation de l'expression qui suit l'invite : on obtient le **résultat réel** (pour l'exemple précédent `2.0`) ;
- évaluation de l'expression sur la 2ème ligne : on obtient le **résultat attendu** (ici aussi `2.0`) ;
- comparaison du résultat réel et du résultat attendu, ce qui produit le **verdict** du test.

Le mécanisme qui compare les valeurs réelle et attendue est appelé **oracle**[1].

Il y a 3 verdicts possibles :

- le résultat réel et le résultat attendu sont les mêmes : c'est un **succès**, on dit que le test passe ;
- le résultat réel et le résultat attendu ne sont pas les mêmes : c'est un **échec**, on dit que le test échoue (**failure** en anglais) ;
- une erreur se produit, qui empêche le calcul du résultat réel et/ou attendu : c'est une **erreur**.

Conventions d'affichage de L1test

Dans la fenêtre L1test de Thonny, les conventions sont les suivantes :

- **succès** : la couleur associée est le **vert** et le message indique que le test est OK ;
- **échec/failure** : la couleur associée est le **rouge** et le message indique `test failed` ;
- **erreur** : la couleur associée est aussi le **rouge** et le message indique `test raised exception`, la levée d'exception étant le mécanisme par lequel Python gère les erreurs.

Exemples

Avec un code réduit à `pass`

Si on reprend la fonction `plus_grande_racine` avec son corps réduit à l'instruction `pass`, le code ne contient aucune instruction `return`. Dans ce cas, Python renvoie par convention la valeur spéciale `None`. Pour les 3 tests le résultat réel est donc `None` et les 3 tests échouent avec un message du type:

```
Test failed for: plus_grande_racine(1, 1, 6)
Expected: 2.0, Got: None
```

ce qu'on peut traduire par "la valeur attendue est `2.0` alors que la valeur réelle est `None`".

Avec un code correct

On suppose maintenant qu'on a écrit un corps de fonction correct pour `plus_grande_racine`[2].

Supposons que la docstring contienne le test suivant :

```
$$ plus_grande_racine(1, 1, -6)
2.0
```

[1] On verra plus tard comment se fait exactement cette comparaison en Python.
[2] Un code est correct s'il est conforme à sa spécification, l'énoncé de l'exercice par exemple.

Oups ! Il manque un caractère `$` dans l'invite… ces 2 lignes sont donc considérées comme du texte libre dans une docstring et pas comme un L1test, qui n'est donc pas exécuté.

Supposons maintenant que la docstring contienne le test suivant :

```
$$$ plus_grande_racine(1, 1, -6)
20
```

Cette fois il manque le `.` dans la valeur attendue, le test échoue avec un message en rouge du type :

```
Test failed for: plus_grande_racine(1, 1, 6)
Expected: 20, Got: 2.0
```

Si par inadvertance nous écrivons un test *qui n'a pas de sens*, en utilisant une valeur de test qui viole les conditions d'utilisation de la fonction[3] avec la valeur `0` pour `a` :

```
$$$ plus_grande_racine(0, 1, -6)
2.0
```

La division par `0` produira une erreur et le test est en erreur avec un message en rouge du type :

```
Test raises exception for: plus_grande_racine(0, 1, -6)
Zero division error: float division by 0
```

Les 3 tests précédemment écrits dans la docstring initiale de `plus_grande_racine` passent, avec un message en vert du type :

```
Test OK for: plus_grande_racine(1, 1, -6)
```

Avec un code incorrect

Écrivons maintenant un code faux, par exemple en prévoyant comme valeur de racine $\frac{b-\sqrt{\Delta}}{2a}$ au lieu de $\frac{-b-\sqrt{\Delta}}{2a}$. Parmi les 3 tests initialement prévus, seul le premier test échouera, avec un message en rouge du type :

```
Test failed for: plus_grande_racine(1, 1, 6)
Expected: 2.0, Got: -2.0
```

On note au passage que les 2 autres tests ne détectent pas l'erreur dans le code.

Questions fréquentes

Pourquoi écrire les tests avant le code ?

- pour vérifier qu'on a bien compris le travail que doit effectuer la fonction à coder, d'après sa spécification ;
- parce que si on écrit les tests après avoir écrit le code, on risque de choisir des valeurs attendues en regardant le code, et non la spécification de la fonction. On risque donc d'écrire des tests faux qui valident un code faux ! (et ça n'arrive pas qu'aux débutants !) ;
- les méthodes actuelles de développement logiciel sont des méthodes "test first".

Si un test échoue, c'est que le code de la fonction est faux ?

Soit le code de la fonction est faux, soit le test est mal écrit.

Si tous les tests passent, le code est correct ?

Malheureusement non. Exécuter des tests peut mettre en évidence une erreur dans le code d'une fonction, mais ne peut pas prouver que le code est correct. En effet on peut très rarement tester toutes les valeurs que peuvent prendre les paramètres d'une fonction. Mais des tests au vert permettent d'avoir une plus grande confiance dans la correction du code, surtout si les jeux de tests sont bien choisis.

Comment bien choisir ses jeux de tests ?

C'est un art qui s'acquiert avec l'expérience.

On choisira classiquement des **valeurs représentatives** des différents comportements. Dans le calcul de la plus grande racine on a choisi d'abord un test impliquant un discriminant strictement positif, puis un test impliquant

[3]Une fonction doit par convention être appelée avec des valeurs des paramètres qui satisfont ses conditions d'utilisation.

un discriminant nul. Le dernier test semble un peu redondant avec le premier, même s'il montre que la valeur `0` est acceptable pour `c`. Par contre aucun test n'implique de coefficient flottant, ce qui est dommage.

On choisira aussi des valeurs dites "**aux limites**", c'est à dire qui illustrent les bornes du domaine de la fonction. Si l'énoncé stipule qu'un paramètre x doit être positif, on ne sait pas trop si la valeur `0` est acceptable. Si elle l'est, on choisira cette valeur comme donnée de test.

C'est vraiment utile de tester ?

C'est la méthode de validation de programme la plus utilisée, y compris dans le monde professionnel. Pour vous, c'est une excellente manière de savoir "si vous avez bon" sur ordinateur, l'autre manière étant la relecture de code.

Peut-on toujours tester une fonction ?

Avec L1test, non. On verra par exemple dans le chapitre suivant comment tester une fonction qui a un comportement aléatoire. D'autres bibliothèques de test plus avancées existent, enseignées en Licence et Master mention Informatique.

Et si les calculs sont approchés ?

Dans le jeu de tests de `plus_grande_racine` on aurait pu utiliser des coefficient à valeurs réelles, par exemple `plus_grande_racine(1.5, 1, -5.2)`. Dans ce cas ce n'est pas facile de prévoir le résultat de la fonction de tête, mais on peut dérouler les calculs du discriminant et des racines dans l'interpréteur Python. On s'aperçoit alors que le résultat est `1.558168127881481`, ce qui n'est pas très illustratif. Comme d'une manière générale les calculs sur flottants sont approchés, il est problématique de prévoir un résultat exact. On peut alors utiliser un arrondi dans les tests impliquant un résultat de type flottant. Par exemple :

```
$$$ round(plus_grande_racine(1.5, 1, -5.2), 2)
1.56
```

Memento

- la spécification des fonctions est complétée par ses contraintes d'utilisation
- la chaîne de documentation `dosctring` comporte une section `Contraintes`
- un jeu de tests est un ensemble de tests incluant appel de fonction sur des données de test et résultat attendu de l'exécution de l'appel
- la chaîne de documentation `dosctring` inclut un jeu de tests dans une section `Exemples`
- selon le résultat de son exécution, un test peut être un succès, un échec ou être en erreur
- des tests qui passent augmentent la confiance qu'on peut avoir dans son code, mais il peut y subsister des erreurs non détectées
- l'outil `L1test` permet d'automatiser et systématiser l'exécution des tests unitaires

Exercices niveau débutant

Utilisation et importation de fonction

Devinez le résultat

21. Dans un premier temps, anticipez la valeur prise par les variables à l'issue de l'exécution de chacune des instructions données.
    Puis, vérifierez si la valeur que vous avez proposée est exacte en

    - copiant ces lignes de code dans un programme que vous exécuterez à l'aide de Thonny,
    - consultant les valeurs des variables définies dans le shell de Thonny ou via le panneau des variables.

```
res1 = len('Quelques mots')
res2 = len("Autre exemple !")
res3 = len(21)
res4 = len('21')
res5 = str(21.1)
res6 = len(res5)
res7 = max(12,8, 12.3, 4)
res8 = max(12, 8, 12.3, 4)
res9 = max(len("compare"), len("  compare"), len(" compare"))
resA = min(len("9"), len("1.1"), len("10"), len("-3"))
resB = max(len("9"), len("len('1.1')"), len("11"))
```

Découvrez une fonction

22. À l'aide de la fonction `help()`, consultez la documentation de la fonction `log()` du module `math`.
    Puis, calculez les valeurs des expressions $\log_{10} 100$ et $\ln 2$.

    Vous devez obtenir, respectivement 2 et environ $0,693$.

Calculez avec des fonctions mathématiques

Cercle

23. Écrivez des expressions qui permettent de calculer les valeurs suivantes pour un rayon $R = 3$ cm :

    - circonférence d'un cercle : $2\pi R$
    - aire d'un disque : πR^2

    Vous devez obtenir environ $18,849$ cm pour la circonférence et $28,274$ cm² pour l'aire.

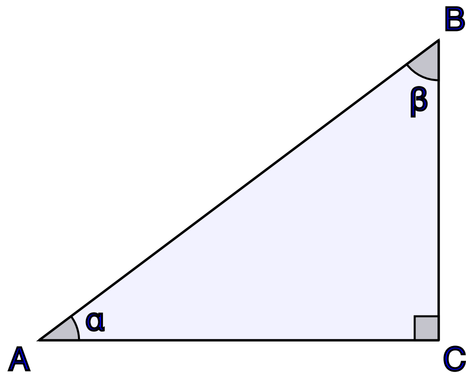
24. Faites de même pour un rayon de 25 m.

    Vous devez obtenir environ 157,079 m pour la circonférence et 1963,495 m² pour l'aire.

Trigonométrie

25. Sachant que $cos\ 180° = cos\ \pi = -1$, testez si les fonctions trigonométriques du module `math` prennent en paramètre des angles en degrés ou en radians.

26. Soit un triangle rectangle illustré ci-dessous, l'égalité suivante s'applique : $cos(\alpha) = \frac{AC}{AB}$



Écrivez une suite d'instructions pour calculer la valeur de AC, sachant que l'angle α mesure 30°, soit $\pi/6$ radians, et que AB mesure 3 cm.

Vous devez obtenir environ $2,598$ cm pour la valeur de AC.

27. Faites de même pour un triangle rectangle où $\alpha = 45° = \pi/4$ et $AB = 11$ cm.

    Vous devez obtenir environ $7,778$ cm pour la valeur de AC.

Utilisez l'aléatoire

28. Générez un nombre entier aléatoire compris entre 1 et 6.
    Exécutez plusieurs fois la ou les instructions pour observer les valeurs obtenues lors d'exécutions successives.
29. Écrivez des instructions afin de choisir au hasard un caractère parmi les suivants : « `+ - / * =` ». De même, exécutez plusieurs fois ces instructions pour observer les différentes valeurs obtenues.

Définition et test de fonction

Documentez une fonction

30. Copiez-collez le code complet – y compris la documentation – de la fonction `celsius2fahrenheit()` vue dans le cours *Définition de fonctions*.
31. Affichez la documentation de cette fonction dans le shell.
32. Sachant que les températures suivantes 30°C, 0°C et -5°C, correspondent, respectivement, à 86°F, 32°F et 23°F, ajoutez trois exemples à la documentation de cette fonction.
33. Ajoutez le code nécessaire pour systématiser les tests unitaires pour cette fonction.

Exercices niveau intermédiaire

Définition et test de fonction

Pour chaque fonction demandée, commencez par écrire la documentation complète, y compris les tests unitaires que vous utiliserez avec le module `doctest`. N'oubliez pas de préciser les contraintes d'utilisation, écrivez `Contrainte : aucune` si vous pensez qu'il n'y en a pas.

Des instructions aux fonctions

Vous allez reprendre les instructions codées dans les exercices précédents de niveau débutant, et les transformer en fonctions.
Vous utiliserez les valeurs données en exercice pour construire vos tests.

Remarque : certains tests unitaires sont fastidieux, voire impossibles pour l'instant, à mettre en place. Lesquels et pourquoi ? Nous verrons dans les prochaines séances comment faire autrement.

34. Écrivez une fonction `cercle_circonference()` qui prend en paramètre la longueur d'un rayon et renvoie la circonférence d'un cercle ayant ce rayon.

35. Écrivez une fonction `cercle_aire()` qui prend en paramètre la longueur d'un rayon et renvoie l'aire d'un cercle ayant ce rayon.

36. Écrivez une fonction `cote_adjacent()` qui prend en paramètre un angle α et la longueur de l'hypoténuse et qui renvoie la longueur du côté adjacent à l'angle α.

Vous pouvez maintenant calculer, par exemple, l'aire d'un cercle avec n'importe quelle valeur de rayon en une seule instruction, alors que vous deviez en écrire plusieurs sans passer par une fonction.

37. Écrivez une fonction `de_6()` qui simule le lancé d'un dé, c'est-à-dire qui ne prend pas de paramètre et qui renvoie un nombre aléatoire compris entre 1 et 6 inclus.

    Appelez plusieurs fois cette fonction `de_6()` pour simuler plusieurs lancés de dés.

    Écrivez une fonction `de()` généralisant la précédente et qui simule le lancé d'un dé dont le nombre de faces est donné en paramètre, et renvoie un nombre aléatoire compris entre 1 et ce nombre de faces inclus.

38. Écrivez une fonction `choisi_caractere()` qui prend en paramètre une chaîne de caractères et qui renvoie un des caractères de cette chaîne choisi aléatoirement.

    Écrivez également plusieurs appels à la fonction `choisi_caractere()` pour observer les valeurs renvoyées.

39. Écrivez une fonction `calcule_interets()` qui reprend la formule de l'exercice de calcul d'intérêts donné dans la feuille d'exercices sur les variables, de niveau intermédiaire. Définissez vous-même quels doivent être les paramètres de cette fonction.

Codez des fonctions simples

40. Écrivez la fonction `addition()` qui prend deux nombres en paramètres et renvoie leur somme.

41. Écrivez la fonction `carre()` qui renvoie le carré d'un nombre passée en paramètre.

42. Écrivez la fonction `cube()` qui renvoie le cube d'une valeur passée en paramètre.

Enluminures

43. Écrivez une série de fonctions `enluminure1()` … `enluminure5()` qui prennent en paramètre une chaîne de caractères quelconque et qui renvoient une autre chaîne (décorative) qui a la forme ci-dessous et ayant un nombre de caractères aussi proche que possible de la longueur de la chaîne passée en paramètre.

    1. de la forme « `---------` »
    2. de la forme « `----O----` »
    3. de la forme « `-o-o-o-O-o-o-o-` »
    4. de la forme « `>>-o-o-o-O-o-o-o-<<` »
    5. de la forme « `<->-<->-<->-O-<->-<->-<->` »

    Par exemple, `enluminure2('Python')` produira la chaîne « `---O---` ».

Utilisez et combinez des fonctions

Dans la console Python, utilisez uniquement les fonctions définies précédemment afin de répondre aux questions suivantes (même les opérateurs arithmétiques sont interdits). Chaque calcul devra être codé en une seule expression.

44. Calculez le carré de $0,01$. Quelle valeur obtenez-vous ? D'après-vous, pourquoi ?
45. Calculez l'aire (ou surface) d'un carré de 4 cm de côté.

    Vous devez obtenir 16 cm^2.
46. Calculez l'aire d'un rectangle de dimensions (en cm) : 4×8. Vous noterez que ce rectangle équivaut à deux carrés de 4 cm de côté, placés côte à côte.

    Vous devez obtenir 32 cm^2.
47. Calculez le volume d'un cube de $5,5$ cm de côté.

    Vous devez obtenir $166,375$ cm^3.
48. Calculez le volume d'un pavé de dimensions (en cm) : $4 \times 4 \times 8$. Vous noterez que ce pavé équivaut à deux cubes de 4 cm de côté, placés côte à côte.

    Vous devez obtenir 128 cm^3.
49. Réalisez deux lancers de dé et additionnez leurs valeurs.

    Il n'est pas possible de savoir quelle valeur sera obtenue, la seule certitude est qu'elle sera comprise entre 2 et 12.
50. Écrivez une instruction qui permet de générer aléatoirement une syllabe composée d'une consonne suivi d'une voyelle.

    Ici aussi, il n'est pas possible d'anticiper la syllabe générée, la seule certitude est que la première lettre sera présente dans la chaîne de caractères *"zrtpqsdfghjklmwxcvbn"* et la deuxième dans la chaîne *"aeiouy"*

En secondes !

Un jour est composé de 24 heures, une heure de 60 minutes, et une minute de 60 secondes.

Sur le modèle suivant

```
def minute2sec(m):
    """Renvoie le nombre de secondes correspondant à un nombre de
    minutes donné.
    Paramètre :
    - m (int) : nombre de minutes à convertir
    Valeur de retour :
    - int : nombre de secondes
    Contraintes : m positif ou nul
    Exemples :
    >>> minute2sec(0)
    0
    >>> minute2sec(42)
```

```
2520
"""
```

51. Proposez trois fonctions
    1. `minute2sec()` de conversion de minutes en secondes
    2. `heure2min()` de conversion d'heures en minutes
    3. `jour2heure()` de conversion de jours en heures
52. Proposez ensuite une fonction de conversion en seconde d'une durée exprimées en jour, heures, minutes secondes.
    Cette fonction `en_secondes()` acceptera donc quatre paramètres : un nombre de jours, un nombre d'heures, un nombre de minutes, et un nombre de secondes.

    On utilisera les fonctions précédemment définies.

Exercices niveau confirmé

Calcul en chaîne

Soit le programme suivant, `calc0.py` :
(*pour les besoins de l'exercices, les fonctions ne comportent pas de docstring*)

```
a = 5
b = 1

def calc0(b, a):
    return (b-a)//2
```

On charge le fichier et on exécute ce programme Python dans l'environnement Thonny.

53. Quelles sont les variables et fonctions définies suite à cette exécution ?

    Vérifiez votre réponse en consultant le volet *Variables* de Thonny.

Suite à cette exécution, on saisit le code suivant dans la console Python (*Shell*).

```
a = calc0(a, b)
```

54. Quelle est alors la valeur associée à la variable `a` ?

On considère maintenant le programme suivant, `calc1.py` :

```
a = 5
b = 1
c = 2

def calc1(a, b):
    return (a+b)//2

def calc2(a, b):
    return (a-b)//2

def calc3(x, y):
    return calc1(x, y) * calc2(y, x)
```

On charge le fichier et on exécute ce programme Python dans l'environnement Thonny.

55. Quelles sont les variables et fonctions définies suite à cette exécution ?

    Vérifiez votre réponse en consultant le volet *Variables* de Thonny.

Suite à cette exécution, on saisit le code suivant dans la console Python (*Shell*).

```
b = calc3(c, a+b)
```

56. Quelle est maintenant la valeur associée à la variable `b` ?

On complète le programme `calc1.py` avec le code suivant :

```
def calc4(b, c):
    x = 1
    y = c-b
    z = calc3(y, 1)
    return calc1(calc2(x,y), z)
```

```
c = calc4(b//2, calc1(a,c))
```

57. Quelle est la valeur associée à chacune des variables `a`, `b`, et `c` suite à une nouvelle exécution du programme ?

Babbage

En 1822, Charles Babbage imagine les plans d'une machine à calculer les valeurs successives d'un polynôme par la méthode des différences, remplaçant ainsi des multiplications et des élévations à une puissance entière par des suites d'additions.

Calcul d'un polynôme de degré 1

Soit à calculer le polynôme $f(x) = 3x + 5$ pour x prenant successivement les valeurs $0, 1, 2, \cdots$.

58. Définissez une fonction `f()` à un paramètre `x` qui renvoie la valeur de ce polynôme en `x`.

    Cette fonction pourra servir à vérifier les calculs menés dans la suite.

Le tableau suivant illustre un calcul possible pour obtenir les valeurs successives de $f(x)$:

- la première colonne contient les valeurs de x,
- la colonne suivante la valeur du polynôme pour x,
- et la troisième colonne les différences entre deux valeurs successives du polynôme.

x	$f(x)$	$g(x) = f(x+1) - f(x)$
0	5	3
1	8	3
2	11	3
3	14	$\cdots$

On définit deux variables et une constante que l'on initialise avec les valeurs de la première ligne de ce tableau :

```
x = 0
fx = 5

GX = 3
```

59. Écrivez une suite d'instructions qui calcule dans `x` et `fx` les valeurs de la ligne suivante.

60. Exécutez ces instructions autant de fois que nécessaire pour compléter le tableau avec les valeurs de $f(4)$ et $f(5)$.

Avec un polynôme de degré 5

Considérons maintenant le polynôme $f(x) = 5x^5 + 2x^4 + 3x^2 + 7x + 1$.

Complétez un peu le tableau suivant :

x	$f(x)$	$g(x) = f(x+1) - f(x)$	$h(x) = g(x+1) - g(x)$	$j(x)$	$k(x)$	$l(x) = k(x+1) - k(x)$	$\cdots$
0	1	17	184	822	1248	600	$\cdots$
1	18	201	1006	2070	1848	600	$\cdots$
2	219	1207	3076	3918	2448	$\cdots$	$\cdots$
3	1426	4283	6994	6366	3048	$\cdots$	$\cdots$
4	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

61. Combien de colonnes utiles contient ce tableau en fonction du degré du polynôme ?

62. Définissez six variables et une constante initialisées avec les valeurs de la première ligne.

63. Donnez la suite d'instructions qui, chaque fois qu'elle sera exécutée, calculera la valeur du polynôme pour la valeur suivante de x

Affectation multiple

Python autorise des affectations multiples.

64. Exécutez les instructions suivantes et observez le résultat à chaque étape :

```
a, b, c = 12, 1, 7
a, b, c = a+b, b+c, c+a
```

Chacune des expressions de la partie droite de l'affectation est évaluée.
Puis, les variables de la partie gauche de l'affectation sont associées aux valeurs produites par ces évaluations.

65. Utilisez cette affectation multiple pour réduire la suite d'instructions de calcul du polynôme précédent à une seule instruction.

3. Type booléen - Expressions booléennes - Prédicats

Expressions booléennes

Coup d'œil

On découvre

- comment manipuler des booléens, les valeurs de vérité *vrai* et *faux*
- comment comparer des nombres ou des chaînes de caractères
- comment construire des expressions booléennes
- comment définir des prédicats, fonctions renvoyant *vrai* ou *faux*

Les booléens

Le type booléen a deux valeurs - *vrai* et *faux* -.

Python définit deux littéraux `True` et `False` pour noter ces valeurs :

```
>>> True
True
>>> type(True)
<class 'bool'>
>>> False
False
```

La casse est signifiante : `True` et `False` s'écrivent avec une majuscule en début de mot.

Une expression dont la valeur ne peut être que vrai ou faux est appelée **expression booléenne**.

Une fonction à valeur booléenne est appelée **prédicat**.

Opérateurs de comparaison

Les opérateurs de comparaison, comme leur nom l'indique, permettent de comparer deux valeurs et de produire une valeur booléenne.

La valeur 12 est inférieure à la valeur 42, vrai ou faux ?

```
>>> 12 < 42
True
```

Python définit les opérateurs suivants :

comparaison	opérateur Python	commentaire
égalité	`==`	à ne pas confondre avec l'affectation `=`
différence	`!=`	l'ordre des caractères est important
inférieur	`<`	
inférieur ou égal	`<=`	l'ordre des caractères est important
supérieur	`>`	
supérieur ou égal	`>=`	l'ordre des caractères est important

Comparer des nombres

Nous pouvons comparer des nombres :

```
>>> a = 6
>>> b = 4
>>> c = 2
>>> a < b
False
>>> a < b+c
False
>>> a == b+c
True
>>> c != a-b
False
```

Nous pouvons comparer des nombres de différents types :

```
>>> 1 + 1 == 2.0
True
```

Comparer des flottants

Les flottants sont des approximations de valeurs entières. La comparaison de flottants nécessite de prendre des précautions.

```
>>> z = 3.0
>>> 3.0 == z
True
```

```
>>> 3 * 0.1 == 0.3
False
>>> 3 * 0.1
0.30000000000000004
```

Nous verrons qu'il est préférable de procéder en vérifiant si l'écart entre deux valeurs est inférieur à un seuil donné à définir, par exemple 0.00000001.

Égalité de chaînes de caractères

Deux chaînes de caractères sont égales si et seulement si :

- elles ont le même nombre de caractères
- le premier caractère de chacune des deux chaînes sont identiques
- le deuxième caractère de chacune des deux chaînes sont identiques
- etc.

On utilise les opérateurs == et != sur les chaînes :

```
>>> 'oui' == 'o' + 'u' + 'i'
True
>>> 'oui' == 'oui '
False
>>> 'oui' != 'Oui'
True
>>> "C cool !" == 'C cool !'
True
>>> "C'est cool !" == 'C\'est cool !'
True
```

Notons que la comparaison de chaînes de caractères est sensible à la casse : les majuscules et minuscules sont des caractères différents.
De plus, l'espace est aussi un caractère.

Par ailleurs, des valeurs littérales différentes permettent d'écrire une même chaîne de caractères (par exemple en utilisant les apostrophes `'` ou les guillemets `"`). Il s'agit cependant bien de la même valeur, la comparaison par == donne vrai.

Prédicats

Un prédicat est donc une fonction qui renvoie une valeur booléenne.

Premier exemple — parité

Définissons un prédicat `est_pair()` pour vérifier la parité d'un entier donné. Voici sa spécification :

```
def est_pair(n):
    """
    Renvoie True si et seulement si n est pair, False sinon.

    Paramètres :
    - n : entier dont on teste la parité
    Valeur de retour :
    - bool
    Contrainte : aucune
    Exemples:
    >>> est_pair(42)
    True
    >>> est_pair(13)
    False
    >>> est_pair(0)
    True
    """
```

Un entier est pair si le reste de sa division entière par 2 est nul.

On pourra donc compléter notre fonction avec le code suivant :

```
def est_pair(n):
    """
    [...]
    """
    return n%2 == 0
```

Vérifions que ce code valide bien les tests :

```
>>> import doctest
>>> doctest.testmod(verbose=True)
Trying:
    est_pair(42)
Expecting:
    True
ok
Trying:
    est_pair(13)
Expecting:
    False
ok
Trying:
    est_pair(0)
Expecting:
    True
ok
1 items had no tests:
    __main__
1 items passed all tests:
   3 tests in __main__.est_pair
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=3)
```

Deuxième exemple — somme paire

Définissons un prédicat qui vérifie si la somme de deux nombres est paire.

Voici une spécification :

```
def somme_est_paire(a, b):
    """
    Renvoie True si et seulement si la somme de deux nombres est paire.

    Paramètres:
    - a : entier
    - b : entier
    Valeur de retour: booléen
    Contrainte : aucune
    Exemples:
    >>> somme_est_paire(21, 42)
    False
    >>> somme_est_paire(42, -2*21)
    True
    >>> somme_est_paire(1, 3)
    True
    """
```

Nous pouvons compléter le code de cette fonction par

```
    somme = a+b
    return somme%2 == 0
```

ou faire appel à notre fonction `est_pair()` :

```
    somme = a+b
    return est_pair(somme)
```

La deuxième solution est à privilégier car elle évite la redondance de code.

Troisième exemple — chaîne non vide

Définissons maintenant un prédicat `est_non_vide()` qui correspond à la spécification suivante :

```
def est_non_vide(str):
    """
    Renvoie True si et seulement si une chaîne de caractères n'est pas vide.
    Paramètre:
    - str : chaîne de caractères
    Valeur de retour: booléen
    Contrainte : aucune
    Exemples:
    >>> est_non_vide('Timoléon')
    True
    >>> est_non_vide('')
    False
    >>> est_non_vide(' ')
    True
    """
```

Le code de cette fonction est par exemple

```
    return str != ''
```

Memento

- le type *booléen* `bool` permet de représenter les valeurs de vérité *vrai* et *faux*
- en Python, on utilise les mots-clés `True` et `False`
- les opérateurs de comparaison testent l'égalité, la différence, l'inférieur, le supérieur de deux valeurs
- un opérateur de comparaison produit une valeur booléenne
- en Python, l'opérateur d'égalité est noté `==`, l'opérateur de non égalité est noté `!=`
- un prédicat est une fonction qui renvoie une valeur booléenne

Opérateurs logiques

Coup d'œil

On découvre

- comment combiner des valeurs booléennes pour en produire d'autres
- les opérateurs logiques *et*, *ou*, *non*
- comment une table de vérité définit un opérateur logique

Opérateurs booléens

Les opérateurs booléens, aussi appelés opérateurs logiques, combinent des valeurs booléennes pour produire une nouvelle valeur booléenne.

Il existe principalement trois opérateurs logiques :

- la négation - le *non* -,
- la conjonction - le *et logique* -,
- la disjonction - le *ou logique* -

La négation – `not`

L'opérateur de *négation* permet d'obtenir la valeur inverse d'une valeur booléenne.

En Python, l'opérateur de négation est noté `not`. On écrit par exemple

```
>>> pair = True
>>> not pair
False
```

ou

```
>>> not 3 < 42
False
>>> not 'Zim' == 'Zac'
True
```

On décrit couramment les opérateurs booléens par leur table de vérité. Pour l'opérateur de négation :

x	not x
False	True
True	False

Le et logique — `and`

Le *et logique* permet d'exprimer le fait que deux expressions sont vraies simultanément : x and y est vrai si et seulement si x est vrai, et y est vrai aussi.

En Python, le *et logique* est noté `and`. On écrit par exemple

```
>>> pair = True
>>> positif = False
>>> pair and positif
False
```

ou

```
>>> a = 12
>>> a < 42 and a > 0
True
>>> 'Zim' != 'Zac' and 'Zim' != 'Zou'
True
```

La table de vérité de l'opérateur *et logique* est la suivante

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Le ou logique — `or`

Le *ou logique* permet d'exprimer le fait qu'une parmi deux expressions est vraie : x or y est vrai si soit x est vrai, soit y est vrai, soit les deux sont vrais.

Remarquons que x or y est aussi vrai si x est vrai et y est vrai.

En Python, le *ou logique* est noté `or`. On écrit par exemple

```
>>> pair = True
>>> positif = False
>>> pair or positif
True
```

ou

```
>>> a = 12
>>> a < 0 or a > 42
False
>>> 'Zim' == 'Zac' or 'Zim' == 'Zou'
False
```

La table de vérité de l'opérateur *ou logique* est la suivante

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Expressions booléennes

Une expression booléenne est une expression à valeur booléenne.

Définissons quelques prédicats qui utilisent des expressions booléennes ou des opérateurs booléens.

Premier exemple – pair positif

Nous avions défini un prédicat `est_pair()` qui vérifie la parité d'un entier donné en paramètre.

Définissons un prédicat `est_pair_positif()` dont voici une spécification :

```python
def est_pair_positif(n):
    """
    Renvoie True si n est pair positif, False sinon.
    Paramètre :
    - n : entier dont on teste la parité et le signe
    Valeur de retour :
    - booléen
    Contraintes : aucune
    Exemples:
    >>> est_pair_positif(42)
    True
    >>> est_pair_positif(-12)
    False
    >>> est_pair_positif(11)
    False
    >>> est_pair_positif(0)
    True
    """
```

Nous pouvons nous baser sur un appel à notre prédicat `est_pair()` et vérifier que le paramètre `n` est positif :

```python
    pair = est_pair(n)
    positif = n >= 0
    return pair and positif
```

ou écrire directement

```python
    return est_pair(n) and n >= 0
```

ou encore

```python
    return n%2 == 0 and n >= 0
```

Second exemple – Zim Zac Zou

Définissons un prédicat qui vérifie qu'une chaîne de caractères est une des deux chaînes « Zim », « Zac » :

```python
def est_zim_zac(str):
    """
    Renvoie True si str est une des chaînes 'Zim', 'Zac'
    Paramètre :
    - str : chaîne de caractères
    Valeur de retour :
    - booléen
    Contraintes : aucune
    Exemples:
    >>> est_zim_zac('Zac')
    True
    >>> est_zim_zac('Zim Zac')
    False
    >>> est_zim_zac('')
    False
    """
```

Il s'agit donc de vérifier si la chaîne est égale à « Zim » : `str == 'Zim'`, *ou* si la chaîne est égale à « Zac » : `str == 'Zac'`.

On peut donc écrire

```python
    return str == 'Zim' or str == 'Zac'
```

Complétons notre prédicat pour vérifier qu'une chaîne de caractères est une des trois chaînes « Zim », « Zac », « Zou ».

Il s'agit de vérifier si la chaîne est égale à une des deux chaînes « Zim », « Zac » (ce que nous venons de faire), *ou* si la chaîne est égale à « Zou ».

On écrira :

```
def est_zim_zac_zou(str):
    """
    Renvoie True si str est une des chaînes 'Zim', 'Zac', 'Zou'
    Paramètre:
    - str : chaîne de caractères
    Valeur de retour :
    - booléen
    Contraintes : aucune
    Exemples:
    >>> est_zim_zac_zou('Zac')
    True
    >>> est_zim_zac_zou('Zim Zac')
    False
    >>> est_zim_zac_zou('')
    False
    """
    return str == 'Zim' or str == 'Zac' or str == 'Zou'
```

Opérateurs booléens séquentiels

Observons la table de vérité de l'opérateur `and` :

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Pour évaluer `x and y` :

- si `x` est évalué à `False`
- quelque soit la valeur de `y`, `x and y` vaudra `False`

Python va utiliser cette remarque pour définir l'opérateur `and`. Ainsi, pour évaluer l'expression `x and y` :

- Python évalue la partie gauche, le `x`
- si elle vaut `False`, on a terminé
- sinon, Python évalue la partie droite, le `y`

On parle d'opérateur **séquentiel** : la partie gauche est évaluée. Puis, si nécessaire, la partie droite est évaluée.

Il en est de même pour l'opérateur `or`.

Memento

- un opérateur logique combine des valeurs booléennes et produit une valeur booléenne
- l'opérateur logique de négation est noté `not` en Python
- l'opérateur *et logique* est noté `and` en Python
- l'opérateur *ou logique* est noté `or` en Python
- on combine ces opérateurs logiques pour former des expressions booléennes
- les opérateurs logiques de Python sont *séquentiels*

Expressions booléennes et doctests

Coup d'œil

On découvre

- l'utilité des expressions booléennes pour écrire certains doctests
- un moyen de tester l'égalité de valeurs flottantes, en particulier dans les doctests
- un moyen de tester l'égalité de chaînes de caractères, en particulier dans les doctests

Expression booléenne

Une expression booléenne est une expression dont la valeur est un booléen.
En Python, ce sera `True` ou `False`.

Vérifier qu'une expression quelconque est égale à un résultat donné peut s'écrire à l'aide d'une expression booléenne.

Par exemple vérifier qu'un appel de la fonction `plus_grande_racine(1, 1, -6)` produit bien la valeur 2 peut s'écrire

```
plus_grande_racine(1, 1, -6) == 2.0
```

On peut aussi désigner la valeur 2 par l'entier Python `2`, et écrire

```
plus_grande_racine(1, 1, -6) == 2
```

Tests de fonctions renvoyant des nombres

Dans la spécification de notre fonction `plus_grande_racine()`, nous écrivions :

```
def plus_grande_racine(a, b, c):
    """
    Renvoie la plus grande racine du polynôme de second degré ax^2 + bx + c

    Paramètres :
    - a : (int or float) coefficient a - associé à x^2
    - b : (int or float) coefficient b - associé à x
    - c : (int or float) coefficient c - constant
    Valeur de retour :
    - float (la plus grande racine du polynôme)
    Contraintes :
    - a doit être non nul
    - (b**2 - 4*a*c) doit être positif ou nul
    Exemples :
    >>> plus_grande_racine(1, 1, -6)
    2.0
    """
```

Mais il nous importe peu que la fonction renvoie `2.0` ou `2`.

On écrira donc notre doctest sous la forme d'une expression booléenne. Par exemple

```
    >>> plus_grande_racine(1, 1, -6) == 2
    True
```

Tests de fonctions renvoyant des réels

Il est impossible de prévoir la valeur exacte d'un réel renvoyé par une fonction à cause des approximations et arrondis.

Il est donc préférable d'utiliser une expression booléenne qui teste un écart avec une valeur attendue.

```
def decimetre2metre(distance):
    """
    Converti en mètres une distance donnée en décimètres
    Paramètre :
    - distance : float une distance en mètres
    Valeur de retour :
    - float - la distance convertie en décimètres
    Contraintes : aucune
    Exemples :
    >>> abs(decimetre2metre(3) - 0.3) < 0.000001
    True
    >>> abs(decimetre2metre(51) - 5.1) < 0.000001
    True
    """
    return distance * 0.1
```

Tests de fonctions renvoyant des chaînes de caractères

Plusieurs valeurs littérales permettent de désigner une même chaîne de caractères, par exemple suivant que l'on utilise des apostrophes `'` ou des guillemets `"` comme délimiteurs.

On préférera ici aussi utiliser une expression booléenne pour tester l'égalité d'une valeur renvoyée avec une chaîne de caractères donnée.

À cette première version :

```
def enlumine(chaine):
    """
    Renvoie une chaîne de caractères ornée de quelques enluminures
    Paramètre :
    - chaine : str - la chaîne à enluminer
    Valeur de retour :
    - str : la chaîne enluminée
    Contraintes : aucune
    Exemples :
    >>> enlumine("bonjour")
    '<<-- bonjour -->>'
    >>> enlumine("j'adore les apostrophes !")
    "<<-- j'adore les apostrophes ! -->>"
    """
    return "<<-- " + chaine +  " -->>"
```

on préfèrera donc

```
def enlumine(chaine):
    """
    Renvoie une chaîne de caractères ornée de quelques enluminures
    Paramètres :
    - chaine : str - la chaîne à enluminer
    Valeur de retour :
    - str : la chaîne enluminée
    Contraintes : aucune
    Exemples :
```

```
>>> enlumine("bonjour") == '<<-- bonjour -->>'
True
>>> enlumine("j'adore les apostrophes !") == "<<-- j'adore les apostrophes ! -->>"
True
"""
return "<<-- " + chaine +  " -->>"
```

Memento

- les doctests peuvent prendre la forme d'expressions booléennes
- il est préférable de passer par une expression booléenne pour tester une fonction qui renvoie un réel
- il est préférable de passer par une expression booléenne pour tester une fonction qui renvoie une chaîne de caractères

Expressions booléennes et L1test

> ## Coup d'œil
>
> On découvre
> - comment L1test compare la valeur attendue et la valeur obtenue d'un test
> - l'utilité des expressions booléennes pour écrire certains L1tests
> - un moyen de tester les fonctions utilisant l'aléatoire
> - un moyen de tester les fonctions renvoyant une valeur flottante

Expression booléenne et évaluation d'un L1test

On rappelle que dans sa forme la plus simple un l1test s'écrit sur 2 lignes :

```
$$$ <expression_à_évaluer>
<expression_résultat_attendu>
```

Pour produire le verdict associé au test, L1test compare la valeur obtenue (par évaluation de `<expression_à_evaluer>`) et la valeur attendue (obtenue par évaluation de `<expression_résultat_attendu>`). Le résultat de la comparaison est vrai (`True`) ou faux (`False`)[4] : la comparaison peut s'écrire par une expression booléenne. L1test procède en utilisant l'opérateur de comparaison `==`.

Exemples

Avec un test correct

Pour évaluer le test suivant :

```
$$$ plus_grande_racine(1, 1, -6)
2.0
```

L1test va évaluer l'expression `plus_grande_racine(1, 1, -6)` (valeur obtenue : `2.0`), l'expression `2.0` (valeur attendue : `2.0`), puis évaluer l'expression booléenne `2.0 == 2.0`. Cette expression vaut `True` : le test passe.

L'expression `2.0 == 2` valant de même `True`, le test peut s'écrire aussi :

```
$$$ plus_grande_racine(1, 1, -6)
2
```

La valeur attendue peut être décrite par une expression si on estime que la lisibilité du test s'en trouve accrue :

```python
def duplication(chaine, n):
    '''
    Renvoie la chaine `chaine` répétée `n` fois.

    Paramètres:
    - chaine (str) : la chaîne à répéter
    - n (int) : le nbre de répétition
    Valeur de retour (str) :
    CU : n >= 0
    Exemples:
```

[4] s'il ne se produit pas une erreur.

```
    $$$ duplication('oh', 0)
    ''
    $$$ duplication('', 4)
    ''
    $$$ duplication('oh', 1)
    'oh'
    $$$ duplication('oh', 4)
    'oh' + 'oh' + 'oh' + 'oh'
    '''
```

NB on peut aussi utiliser les différentes syntaxes des littéraux de type chaîne. À la place de la valeur attendue `'oh'` on aurait pu utiliser `"oh"`, car l'expression booléenne `'oh' == "oh"` vaut `True`.

Avec un test incorrect

Pour évaluer le test suivant :

```
$$$ plus_grande_racine(1, 1, -6)
20
```

L1test va évaluer l'expression `plus_grande_racine(1, 1, -6)` (valeur obtenue : `2.0`), l'expression `20` (valeur attendue : `20`), puis évaluer l'expression booléenne `2.0 == 20`. Cette expression vaut `False` : le test est en échec.

Tests de fonctions utilisant l'aléatoire

Quand le résultat d'une fonction dépend d'un calcul qui utilise l'aléatoire, on ne peut pas prévoir ce résultat.

Considérons la fonction `de_6` qui simule et renvoie le lancer d'un dé à 6 faces.

```
def de_6():
    """
    Renvoie une valeur entière tirée aléatoirement entre 1 et 6.

    Paramètre : aucun
    Valeur de retour (int) : une valeur entière entre 1 et 6
    Contraintes : aucune
    Exemples : CI DESSOUS
    """
    return random.randint(1,6)
```

Tout ce qu'on peut dire du résultat d'un appel à `de_6`, c'est que la valeur obtenue est comprise entre `1` et `6`. C'est la description d'une propriété du résultat. Cette propriété, dans notre cas un encadrement, peut être décrite par une expression booléenne. On écrit ainsi les tests :

```
$$$ 1 <= de_6()
True
$$$ de_6() <= 6
True
```

Cette forme de tests ne signifie pas que la fonction `de_6` est un prédicat : la fonction renvoie bien un entier. Elle signifie que, à défaut de pouvoir décrire exactement son résultat, on décrit une propriété booléenne de ce résultat.

NB il doit être bien clair que si on écrit le jeu de tests incorrect suivant :

```
$$$ de_6()
1
$$$ de_6()
2
[...]
$$$ de_6()
5
$$$ de_6()
6
```

alors, chaque test effectuant son propre appel à la fonction `de_6`, peut-être que certains tests passeront, peut-être qu'aucun test ne passera.

Si on souhaite énumérer toutes les valeurs possibles, on peut écrire cette forme particulière de l1test :

```
$$$ v = de_6()
$$$ v == 1 or v == 2 or v == 3 or v == 4 or v == 5  or v == 6
True
```

La première ligne est une affectation : `v` prend la valeur d'un appel à `de_6`. Il n'y a pas de verdict, pas de comparaison effectuée. La seconde ligne est un l1test classique qui passe ssi la valeur de `v` est bien comprise entre 1 et 6. Il y aura un seul verdict d'affiché dans la fenêtre de test.

NB De même le test suivant est incorrect

```
$$$ de_6()
1 or 2 or 3 or 4 or 5 or 6
```

car même si aucune erreur de syntaxe n'est levée, l'expression décrivant la valeur attendue n'est pas une expression booléenne correctement construite (la valeur de l'expression[5] `1 or 2 or 3 or 4 or 5 or 6` vaut `1`). Comme le précédent, ce test passera ou échouera au gré des exécutions.

Tests de fonctions renvoyant des réels

Il est impossible de prévoir la valeur exacte d'un réel renvoyé par une fonction à cause des approximations et arrondis. Par exemple :

```
>>> 3 * 0.1
0.30000000000000004
```

Ainsi, pour tester la fonction suivante :

```python
def decimetre2metre(distance):
    """
    Convertit en mètres une distance donnée en décimètres.

    Paramètre :
    - distance (float): une distance en mètres
    Valeur de retour (float) : la distance convertie en décimètres
    Contraintes : distance >= 0
    Exemples : CI DESSOUS
    """
    return distance * 0.1
```

on ne pourra pas écrire que la valeur attendue pour `decimetre2metre(3)` est `0.3`, puisque ce n'est pas le cas.

On pourra écrire le test en utilisant une expression booléenne qui évalue la valeur absolue de l'écart avec la valeur attendue :

```
$$$ abs(decimetre2metre(3) - 0.3) < 0.000001
True
```

À nouveau, ça ne veut pas dire que `decimetre2metre` est un prédicat : `decimetre2metre` renvoie un flottant. Mais le test décrit une propriété booléenne du résultat renvoyé.

Ou encore on pourra utiliser la fonction Python prédéfinie `isclose`, qui prend en paramètre deux nombres et qui renvoie `True` si ces deux nombres sont proches, avec une tolérance de 10^{-9}. Un troisième paramètre optionnel peut être ajouté pour préciser la tolérance :

```
$$$ isclose(decimetre2metre(3), 0.3)
True
$$$ isclose(decimetre2metre(561), 56, rel_tol=0.01)
True
```

[5] La manière dont s'évalue cette expression n'est pas traitée dans le cours.

Memento

- les tests peuvent prendre la forme d'expressions booléennes
- il est nécessaire de passer par une expression booléenne pour tester une fonction qui utilise l'aléatoire
- il est possible de passer par une expression booléenne pour tester une fonction qui renvoie un réel

Exercices niveau débutant

Expressions booléennes

Pour les questions qui suivent, n'utilisez pas tout de suite l'interpréteur Python.

- Anticipez les valeurs des expressions booléennes (`True` ou `False`).
- Vérifiez, ensuite la validité de votre réponse en utilisant l'interpréteur.

Opérateurs de comparaison

66. ```
    x = 3
    12 < x
    x == 1+2
    5 != x
    x + 5. >= x
    0.1 == 1/10
    "test" == "Test"
    "test" == "t" + 'est'
    'Test ' != "Test "
    "Test" != " Test"
    "1" + "2" == "12"
    ```

Opérateurs booléens

67. Quelques expressions booléennes à étudier :

    ```
    not True
    x = 6
    x > 5 and x < 10
    x > 5 or x < 10
    x = 5
    x > 5 and x < 10
    x > 5 or x < 10
    ```

68. En déduire une expression pour vérifier si un nombre appartient à un intervalle, exemple $x \in]5, 10[$
69. Et une expression qui vérifie si un nombre n'appartient pas à un intervalle, exemple $x \notin]5, 10[$

Prédicats

Il s'agit de définir des prédicats (fonctions renvoyant un booléen). Pour chacun :

- on définira quels sont ses paramètres,
- on définira quelles sont les contraintes d'utilisation, s'il y en a,
- on rédigera une docstring complète, y compris des doctests,
- on écrira le code la fonction (pour ces exercices, c'est toujours possible en une seule instruction),
- on validera ce code à l'aide des doctests.

70. Proposez des prédicats pour vérifier :
    1. si un nombre passé en paramètre est impair
    2. si un nombre passé en paramètre est multiple de 7
    3. si une chaîne de caractères est vide (proposez deux codes différents)
    ```

4. si un nombre est positif
5. si deux nombres sont tous deux positifs
6. si deux nombres sont de même signe
7. si deux nombres sont de signes opposés
8. si un nombre donné est dans un intervalle donné (bornes comprises).

États de l'eau

À une pression atmosphérique normale, l'eau est à l'état solide lorsque la température est inférieure à 0°C, à l'état gazeux au delà de 100°C, et à l'état liquide sinon. La température ne peut atteindre la valeur -273.15°C et les valeurs inférieures.

71. Proposez un prédicat `est_glace()` pour vérifier si l'eau est sous forme de glace à une température donnée en paramètre.

72. Proposez un prédicat `est_liquide()` pour vérifier si l'eau est liquide à une température donnée.

Tolérance pour les flottants

Les approximations de la représentation des flottants empêchent l'utilisation de l'égalité :

```
>>> 0.3 == 3 * 0.1
False
```

L'égalité sur les flottants est donc avantageusement remplacée par une comparaison à ϵ près.

73. Proposez un prédicat qui vérifie si deux valeurs flottantes sont proches. En sus des deux valeurs, le prédicat acceptera un troisième paramètre représentant la tolérance acceptable.

Remarque : Le module `math` de la bibliothèque Python fournit un prédicat `isclose()` qui vérifie si deux valeurs sont proches l'une de l'autre ou non, prédicat que l'on utilisera pas ici !

Tests de fonctions

Reprenez certaines des fonctions codées dans les exercices sur les fonctions et réécrivez leurs doctests en utilisant une expression booléenne.

Attention,

- pour les fonctions qui renvoient un nombre à virgule flottante, on prendra en compte que cette valeur est une approximation,
- pour les fonctions qui renvoient une valeur aléatoire, on vérifiera par exemple que la valeur est dans un intervalle donné.

Les fonctions à modifier sont :

1. `celsius2fahrenheit()`
2. `cercle_circonference()`
3. `de_6()`
4. `de(nfaces)`
5. `enluminure1()`

Exercices niveau intermédiaire

Prédicats

Pour chacun des prédicats (fonctions renvoyant un booléen),

- on définira quels sont ses paramètres,
- on définira quelles sont les contraintes d'utilisation, s'il y en a,
- on rédigera une docstring complète, y compris des doctests,
- on écrira le code la fonction (pour ces exercices, c'est toujours possible en une seule instruction),
- on validera ce code à l'aide des doctests.

Lorsque c'est possible, vous utiliserez les prédicats définis dans les exercices précédents (y compris ceux du niveau débutant) pour écrire le code des nouveaux prédicats.

Intervalles

Étant donnés deux intervalles passés en paramètres, écrivez un prédicat pour vérifier si

74. leur intersection est vide
75. le second est inclus dans le premier
76. ils se recouvrent (*ie* le premier inclus dans le second ou inversement)

Quatre vingt-et-un

L'objectif est de déterminer si un lancer de trois dés six (dés à 6 faces) forme un 421. On procédera par étapes.

77. Proposez un prédicat qui renvoie vrai si et seulement si (ssi) deux dés forment un 42, c'est-à-dire que l'un des dés est un 2, l'autre un 4.

```
def est_un_42(de1, de2):
    """
    Renvoie vrai si et seulement si deux dés forment un 42
    Paramètres :
    - de1 : entier - un lancer de dé six
    - de2 : entier - un autre lancer de dé six
    Valeur de retour : booléen
    Contraintes : de1 et de2 appartiennent à [1, 6]
    :Exemples:
    >>> est_un_42(4, 2)
    True
    >>> est_un_42(2, 4)
    True
    >>> est_un_42(4, 4)
    False
    >>> est_un_42(5, 1)
    False
    """
```

78. Proposez un prédicat qui renvoie vrai si et seulement trois dés forment un 421. On utilisera le prédicat précédent.

79. Proposez maintenant un prédicat qui n'utilise pas le prédicat qui teste le 42. Il énumérera donc les différentes combinaisons possibles.

En automne

80. Considérant que l'automne débute le 22 septembre et termine le 20 décembre, proposez un prédicat qui renvoie vrai si et seulement si une date donnée est en automne. Complétez :

```python
def en_automne(jour, mois):
    """Renvoie vrai si et seulement si une date donnée est en
    automne.
    Paramètres :
    - jour (int) : numéro du jour
    - mois (int) : numéro du mois
    Valeur de retour :
    - bool
    Contraintes :
    - 1 <= mois <= 12 et 1 <= jour <= njour
      njour étant le nombre de jours du mois (31 pour janvier...)
    Exemples :
    >>> en_automne(1, 1)
    False
    >>> en_automne(21, 9)
    False
    >>> en_automne (22, 9)
    True
    """
```

Au cinéma

Au cinéma, le tarif réduit est accordé

- aux mineurs - moins de 18 ans
- aux seniors - plus de 60 ans

Nous désirons savoir si - étant donnée sa date de naissance - une personne peut bénéficier du tarif réduit.

Nous allons procéder par étapes.

81. Proposez un prédicat pour comparer deux dates fournies sous la forme de 3 entiers *yyyy*, *mm*, *jj*. Ce prédicat acceptera donc 6 paramètres et renverra `True` quand la première date est antérieure à la seconde.

On suppose la date du jour représentée par trois constantes entières, connues des prédicats. Par exemple

```python
TODAY_YEAR = 2020
TODAY_MONTH = 4
TODAY_DAY = 1
```

82. Proposez un prédicat qui indique si, étant donné sa date de naissance, une personne est senior.

83. Proposez un prédicat qui indique si, étant donné sa date de naissance, une personne est mineure.

De plus, le tarif réduit est également accordé

- aux étudiants de moins de 26 ans

84. Proposez un nouveau prédicat pour déterminer si un client ou une cliente bénéficie du tarif réduit. Ce prédicat prend en paramètre une date de naissance et un booléen qui indique si la personne est étudiante.

Autour du calendrier

Une première fonction de manipulation de dates et de calendriers que nous compléterons par la suite.

Année bissextile

L'année compte habituellement 365 jours. Les années bissextiles comptent 366 jours.

Depuis l'ajustement du calendrier grégorien, l'année est bissextile :

- si l'année est divisible par 4 et non divisible par 100, ou
- si l'année est divisible par 400.

Ainsi, 2019 n'était pas bissextile.
L'année 2020 est bissextile suivant la première règle : divisible par 4 et non divisible par 100.
L'an 1900 n'était pas bissextile car divisible par 4 mais aussi par 100, et non divisible par 400.
L'an 2000 était bissextile car divisible par 400.

(Source Wikipédia `fr.wikipedia.org/wiki/Année_bissextile`)

85. On pourra définir un prédicat préliminaire de test de divisibilité :

```python
def est_divisible_par(a, b):
    """
    Test de divisibilité de deux entiers.
    Paramètres :
    - a, b : deux entiers
    Valeur de retour : booléen
    - True si a est divisible par b
    - False sinon
    Contraintes :
    - b non nul
    Exemples:
    >>> est_divisible_par(10, 2)
    True
    >>> est_divisible_par(10, 3)
    False
    """
```

86. Définissez un prédicat `est_bissextile()` :

```python
def est_bissextile(annee):
    """
    Paramètres :
    - annee : entier - l'année à tester
    Valeur de retour : booléen
    - True si annee est une année bissextile
    - False sinon
    Contraintes :
    - annee > 1582, année d'établissement du calendrier grégorien
    Exemples :
    >>> est_bissextile(2019)
    False
    >>> est_bissextile(2020)
    True
    >>> est_bissextile(1900)
    False
    >>> est_bissextile(2000)
    True
    """
```

Exercices niveau confirmé

Ou exclusif - `xor`

Le *ou exclusif* - noté *xor* - est un autre opérateur booléen. Il permet d'exprimer le fait qu'une et une seule valeur parmi deux est vraie : x xor y est vrai si soit x est vrai, soit y est vrai, mais pas les deux ensemble. La table de vérité de l'opérateur *ou exclusif* est donc la suivante

x	y	x xor y
False	False	False
False	True	True
True	False	True
True	True	False

87. Observez les couples de valeurs de `x` et de `y` pour lesquels `x xor y` est vrai.
88. Écrivez le prédicat `xor()` qui prend en paramètre deux booléens et qui renvoie `True` si l'un ou l'autre des paramètres est `True`.

    Vous devez écrire tous les tests unitaires pour ce prédicat puisqu'il est possible d'énumérer toutes les paires possibles de paramètres.

Intersection et recouvrement de rectangles

La figure ci-dessous illustre un rectangle défini par deux points (xmin, ymin) et (xmax, ymax) dans un repère orthonormé.

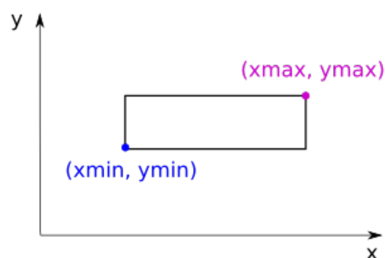


Figure 2: Rectangle dans un repère orthonormé

89. Écrivez un prédicat qui prend en paramètre les coordonnées de ces 2 points et celles d'un troisième point et qui renvoie `True` si le troisième point est inclus dans le rectangle.
90. Écrivez un prédicat qui vérifie si le premier rectangle passé en paramètre est inclus dans le second.
91. Écrivez un prédicat qui vérifie si deux rectangles ont une intersection vide.

Les lois de De Morgan

Les descriptions proviennent de la page Wikipédia sur les Lois de De Morgan.

Les lois de De Morgan sont des identités entre propositions logiques. Elles ont été formulées par le mathématicien britannique Augustus De Morgan (1806-1871).

1re loi — négation de la disjonction La négation de la disjonction de deux propositions est équivalente à la conjonction des négations des deux propositions.
Cela signifie que « not(A or B) » est équivalent à « not(A) and not(B) ».

2nde loi — négation de la conjonction La négation de la conjonction de deux propositions est équivalente à la disjonction des négations des deux propositions.
Cela signifie que « not(A and B) » est équivalent à « not(A) or not(B) ».

Nous allons écrire une série de prédicats afin de démontrer ces lois.

92. Définissez une fonction `de_morgan1()` qui accepte deux valeurs boolénnes en paramètre et renvoie le résultat de la comparaison entre les deux membres de l'équivalence de la 1re loi pour ces valeurs.

93. La première loi de De Morgan est vraie si la fonction `de_morgan1()` renvoie vrai pour toute combinaison de paramètres.

    Définissez une fonction `loi_de_de_morgan1()` qui renvoie vrai si et seulement si chacun des appels possibles à `de_morgan1()` renvoie vrai :

    - remarquez que la fonction `de_morgan1()` acceptant deux paramètres, chacun des paramètres pouvant prendre deux valeurs, il existe quatre combinaisons possibles des paramètres.
    - la fonction `loi_de_de_morgan1()` n'attend pas de paramètre. Si un appel de la fonction renvoie vrai, la 1re loi de De Morgan est prouvée.
      La documentation de la fonction comportera donc un unique doctest :

      ```
      >>> loi_de_de_morgan1()
      True
      ```

94. Définissez maintenant deux fonctions `de_morgan2()` et `loi_de_de_morgan2()` pour vérifier la seconde loi de De Morgan.

Complétude du *non-et*

Opérateur *non-et*

Le *non-et* ou *nand* est un autre opérateur booléen défini par la table de vérité suivante :

x	y	x nand y
False	False	True
False	True	True
True	False	True
True	True	False

L'expression x nand y est vraie si et seulement si au moins un des deux opérandes est faux.

95. Proposez une fonction `nand()` correspondant à cet opérateur.

Complétude du *non-et*

L'opérateur *nand* est un opérateur *complet*. Cela signifie que toute expression de l'algèbre de Boole peut être écrite uniquement à l'aide *nand*.

Afin de confirmer cette propriété, nous allons définir des fonctions pour écrire les trois opérateurs *not*, *and*, et *or* à l'aide de l'unique *nand*.

96. Opérateur *not*
    1. Quelle est la valeur de `nand(True, True)` ?
       De `nand(False, False)` ?
    2. En déduire une expression équivalente à « not x » n'utilisant que l'opérateur *nand*.
    3. Proposez une fonction `not_with_nand()` équivalente au *not* écrite à base de seuls appels à `nand()`.

97. Opérateur *and*
    1. Comparez les tables de vérités des opérateurs *and* et *nand*.
    2. En déduire une expression équivalente à « x and y » n'utilisant que l'opérateur *nand*.
    3. Proposez une fonction `and_with_nand()` équivalente au *and* écrite à base de seuls appels à `nand()`.

98. Opérateur *or*

    1. Proposez une expression équivalente à x or y » n'utilisant que l'opérateur *nand*.
    2. Proposez une fonction `or_with_nand()` équivalente au *or* écrite à base de seuls appels à `nand()`.

99. Proposez enfin un prédicat `verification_completude_nand()` qui renvoie vrai si et seulement si l'ensemble des comparaisons suivantes sont vraies :

    - `not_with_nand(a) == not a` pour toutes les valeurs de `a` possibles
    - `and_with_nand(a, b) == a and b` pour tous les couples de valeurs de `a` et `b` possibles
    - `or_with_nand(a, b) == a or b` pour tous les couples de valeurs de `a` et `b` possibles

Porte logique NAND

Les opérations arithmétiques exécutées par les processeurs sont réalisées à base de circuits. Ces circuits sont construits à base de portes logiques. Une porte logique étant une réalisation matérielle – électronique – d'un opérateur logique.

La complétude de l'opérateur *nand* fait qu'il est possible de construire un processeur à l'aide de seules portes logiques NAND.

De plus, la porte logique NAND est particulièrement facile à construire, les processeurs courants sont effectivement construits à base de cette unique porte.

4. Instruction conditionnelle

Instruction conditionnelle

Coup d'œil

On découvre

- comment exécuter certaines parties d'un programme dans certaines situations
- ce qu'est une instruction conditionnelle

Instruction conditionnelle

Dans certaines situations, on peut vouloir n'exécuter une instruction – ou une suite d'instructions – que si une condition est vérifiée.

Par exemple, corriger le nombre de jours du mois de février dans le cas où l'année est bissextile. Autre exemple, ne mener le calcul des racines d'une équation du second degré que si le déterminant est positif.

Il nous faut pourvoir identifier

- la condition : ce sera une valeur booléenne
- la ou les instructions à exécuter : ce sera un bloc d'instructions, repéré par une indentation supplémentaire

Instruction `if`

En Python, on utilisera le mot clé `if` et on écrira par exemple :

```
njours = 28
if est_bissextile(annee) :
    njours = 29
```

ou

```
if delta >= 0 :
    racine1 = (-b + sqrt(delta)) / (2 * a)
    racine2 = (-b - sqrt(delta)) / (2 * a)
```

Notez le caractère `:` qui termine la première ligne et introduit le bloc d'instructions.

La fin du bloc de code est marquée par la reprise de l'indentation précédente. Et donc par exemple, si notre calcul du nombre de jours était suivi d'une instruction d'affectation pour le calcul du nombre d'heures, on pourrait écrire

```
njours = 28
if est_bissextile(annee) :
    njours = 29
nheures = 24 * njours
```

Flot d'exécution

Le flot d'exécution d'un programme est l'ordre dans lequel les instructions sont exécutées.

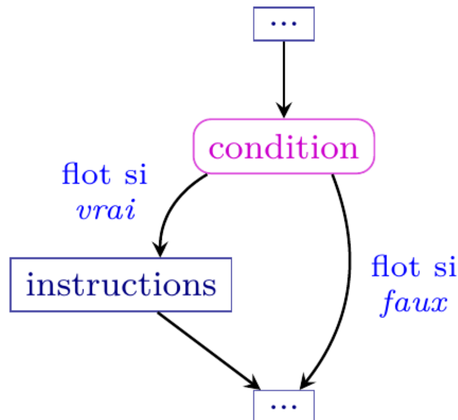
Ce flot d'exécution suit les instructions d'une séquence les unes après les autres.
(*Note à propos de ce flux d'exécution : nous avons vu qu'un appel de fonction permet 1.- d'interrompre la séquence pour exécuter les instructions de la fonction et, 2.- de reprendre ensuite l'exécution là où elle avait été laissée.*)

Plutôt que d'exécuter les unes après les autres les instructions d'un programme, l'instruction conditionnelle `if` permet de potentiellement modifier ce flot d'exécution.

Considérons le code suivant :

```
...
if condition :
    instructions
...
```

Le flot d'exécution correspondant peut être illustré par le graphe :



Dans l'ordre :

- la condition est évaluée. On obtient une valeur booléenne.
- si cette valeur est *vrai*
  - les instructions associées au `if` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (...).
- si cette valeur est *faux*
  - on passe directement à l'exécution des instructions suivantes (...).

Imbrications

Le bloc d'instructions d'un `if` peut lui-même comporter d'autres instructions conditionnelles.
Une instruction `if` peut elle-même être placée dans un autre bloc, par exemple le bloc d'instructions d'une fonction.

Nous avons alors à faire avec des blocs d'instructions *imbriqués*.

La manière dont les instructions sont indentées contrôle cette imbrication. Il faut y être très attentif.

Dans le code suivant

```
if mois == "février" :
    njours = 28
    if est_bissextile(annee) :
        njours = 29
    nheures = 24 * njours
```

l'instruction de calcul de `nheures` fait partie du bloc d'instructions du `if mois == "février"`, elle ne sera donc exécutée que si le mois est février.

Le code suivant diffère très peu du précédent

```
if mois == "février" :
    njours = 28
    if est_bissextile(annee) :
        njours = 29
nheures = 24 * njours
```

L'instruction de calcul du nombre d'heures sera ici exécutée, que le mois soit février ou pas.

Observons aussi l'indentation des instructions de cette fonction (la docstring est abrégée pour les besoins de la présentation) :

```
def nombre_heures(mois, annee) :
    """Renvoie le nombre d'heures d'un mois donné du 1er trimestre."""
    if mois == "janvier" or mois == "mars" :
        njours = 31
    if mois == "février" :
        njours = 28
        if est_bissextile(annee) :
            njours = 29
    nheures = 24 * njours
    return nheures
```

Le code la fonction contient quatre instructions :

- *lignes 3-4* : une première instruction conditionnelle, `if mois == "janvier" or ...`
- *lignes 5-8* : une seconde instruction conditionnelle, `if mois == "février"`

  Le bloc d'instructions de cette conditionnelle est lui-même formé de deux instructions. La seconde étant une instruction conditionnelle.
- *ligne 9* : l'instruction de calcul de `nheures`
- *ligne 10* : l'instruction de terminaison de la fonction.

Cette lecture de la structure du code se fait sur la base de l'indentation des instructions.

Memento

- l'instruction conditionnelle `if` permet de n'exécuter un bloc d'instructions que si une condition est vraie
- les instructions conditionnelles peuvent être imbriquées
- l'indentation définit la manière dont les instructions seront exécutées. Le soin à apporter à cette indentation est capital.

Instruction conditionnelle et alternative

Coup d'œil

On découvre

- comment exécuter certaines parties d'un programme dans certaines situations, et d'autres parties du programme dans les autres situations
- qu'une alternative peut être ajoutée à une instruction conditionnelle

Alternative

L'instruction conditionnelle `if` permet d'exécuter un bloc d'instructions si et seulement si une condition est vraie.
Il existe d'autres situations où l'on peut vouloir exécuter un bloc d'instructions si une condition est vérifiée, et exécuter un autre bloc d'instructions si cette même condition n'est pas vérifiée.

Par exemple, définir le nombre de jours du mois de février à 29 dans le cas où l'année est bissextile, et définir ce nombre de jours à 28 dans les autres cas.
Ou par exemple mener le calcul des racines d'une équation du second degrés d'une certaine manière dans le cas où le coefficient a est nul, et d'une autre manière dans les autres cas.

Il nous faut pouvoir identifier

- la condition : ce sera une valeur booléenne
- la ou les instructions à exécuter quand la condition est vraie : ce sera un bloc d'instructions
- la ou les instructions à exécuter quand la condition n'est pas vraie : ce sera un autre bloc d'instructions

Mot-clé `else`

En Python, on utilisera le mot clé `else`, en association au mot clé `if` que nous connaissons. On écrira par exemple :

```
if est_bissextile(annee) :
    njours = 29
else :
    njours = 28
```

Notez le caractère `:` qui suit le `else` et introduit le bloc d'instructions à exécuter dans le cas où la condition n'est pas vraie.

Notez aussi que la ligne comportant le mot clé `else` est indentée au même niveau que celle du mot clé `if`.

Comme toujours, la fin des blocs de code est marquée par la reprise de l'indentation précédente. Et donc par exemple, si notre calcul du nombre de jours était suivi d'une instruction d'affectation pour le calcul du nombre d'heures, on pourrait écrire

```
if est_bissextile(annee) :
    njours = 29
else :
```

```
    njours = 28
nheures = 24 * njours
```

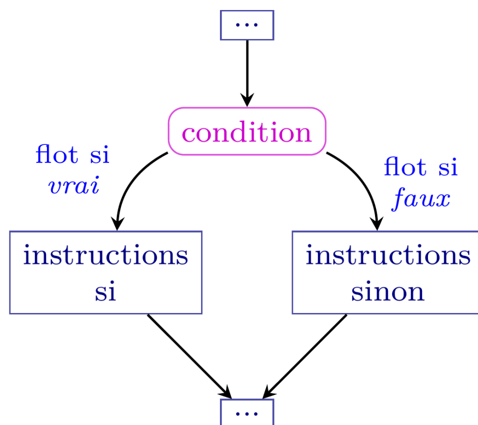
Flot d'exécution

La construction `if … else` permet donc de diriger le flot d'exécution vers un bloc d'instructions ou un autre, en fonction d'une condition.

Le flot d'exécution correspondant au code suivant :

```
...
if condition :
    instructions si
else :
    instructions sinon
...
```

peut être illustré par :



L'exécution consistera, dans l'ordre

- la condition est évaluée. On obtient une valeur booléenne.
- si cette valeur est *vrai*
  - les instructions associées au `if` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (. . .)
- si cette valeur est *faux*
  - les instructions associées au `else` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (. . .)

Notre exemple

```
if est_bissextile(annee) :
    njours = 29
else :
    njours = 28
```

peut être traduit en

```
Si annee est bissextile
    on affecte la valeur 29 à njours
Sinon, c'est-à-dire si annee n'est pas bissextile
    on affecte la valeur 28 à njours
```

Imbrication

Les blocs d'instructions conditionnels et alternatifs peuvent eux-mêmes contenir des instructions conditionnelles :

```
# nombre de jours et d'heures d'un mois donné du 1er trimestre
if mois == "janvier" or mois == "mars" :
    njours = 31
```

```
else:  # février
    if est_bissextile(annee) :
        njours = 29
    else :
        njours = 28
nheures = 24 * njours
```

Bien évidement, la bonne indentation des instructions est essentielle. Comparez les deux codes :

```
# nombre de jours et d'heures d'un mois donné du 1er trimestre
if mois == "février" :
    njours = 28
    if est_bissextile(annee) :
        njours = 29
else : # janvier ou mars
    njours = 31
nheures = 24 * njours
```

et

```
# nombre de jours et d'heures d'un mois donné du 1er trimestre
# version erronée
if mois == "février" :
    njours = 28
    if est_bissextile(annee) :
        njours = 29
    else : # janvier ou mars
        njours = 31
nheures = 24 * njours
```

Dans ce dernier exemple, le `else` de la ligne 7 se rapporte au `if` de la ligne 5, et ne correspond donc pas aux mois de janvier ou mars, mais aux mois de février des années non bissextiles...

Memento

- l'alternative `else` permet de n'exécuter un bloc d'instructions que si la condition d'un instruction conditionnelle `if` est fausse

Instruction conditionnelle - Alternatives multiples

Coup d'œil

On découvre

- comment exécuter certaines parties d'un programme dans différentes situations
- que plusieurs alternatives peuvent être énumérées dans une instruction conditionnelle

Énumération d'alternatives

Nous avons vu que :

- l'instruction conditionnelle `if` permet d'exécuter un bloc d'instructions si et seulement si une condition est vraie,
- l'alternative `else` permet d'exécuter un autre bloc d'instructions si cette condition est n'est pas vraie.

Il existe d'autres situations où l'on peut vouloir exécuter d'autres blocs d'instructions si d'autres conditions sont vérifiées.
Nous sommes face à une série de situations, et pour chacune d'elles voulons exécuter des instructions particulières.

Par exemple, définir le nombre de jours d'un mois du premier semestre qui peut être 31, 30, 28 ou 29 jours.

Il nous faut pouvoir identifier

- une première condition : ce sera une valeur booléenne
- la ou les instructions à exécuter quand cette condition est vraie : ce sera un bloc d'instructions
- une deuxième condition, toujours une valeur booléenne
- la ou les instructions à exécuter quand cette condition est vraie : ce sera un autre bloc d'instructions
- … autant d'autres conditions que nécessaire
- … autant d'autres blocs d'instructions associés à ces conditions
- éventuellement la ou les instructions à exécuter quand aucune de ces conditions n'est vraie : ce sera un dernier bloc d'instructions

Mot-clé `elif`

En Python, on utilisera le mot clé `elif` (pour *else if*), en association au mot clé `if` que nous connaissons. On écrira par exemple :

```
if mois == "janvier" or mois == "mars" or mois == "mai":
    njours = 31
elif mois == "avril" or mois == "juin" :
    njours = 30
else:  # février
    if est_bissextile(annee) :
        njours = 29
    else :
        njours = 28
nheures = 24 * njours
```

Notez la construction identique du `if` et du `elif` avec une condition suivie du caractère : pour introduire un bloc d'instructions à exécuter dans le cas où la condition est vraie.

Notez aussi que les lignes comportant les mot-clés `if`, `elif` et `else` sont indentées au même niveau.

Comme toujours, la fin des blocs de code est marquée par la reprise de l'indentation précédente.

Flot d'exécution

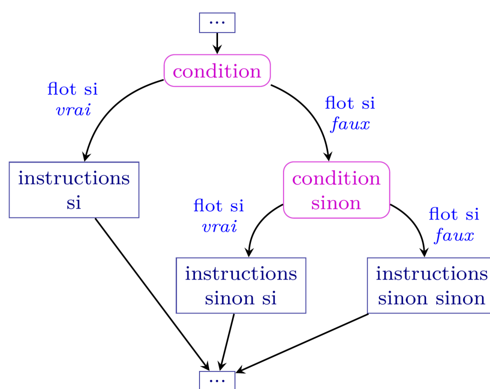
La construction `if` … `elif` … `else` permet donc de diriger le flot d'exécution vers un bloc d'instructions ou un autre, en fonction de différentes conditions.

Notez bien que le flot d'exécution passera par un et un seul des blocs d'instructions.

Le flot d'exécution correspondant au code suivant :

```
...
if condition :
    bloc d'instructions si
elif condition sinon :
    bloc d'instructions sinon si
else :
    bloc d'instructions sinon sinon
...
```

peut être illustré par :



L'exécution consistera, dans l'ordre

- la première condition est évaluée. On obtient une valeur booléenne.
- si cette valeur est *vrai*
  - les instructions associées au `if` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (...)
- si cette valeur est *faux*, la deuxième condition est évaluée. On obtient une nouvelle valeur booléenne.
- si cette nouvelle valeur est *vrai*
  - les instructions associées au `elif` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (...)
- les conditions suivantes sont évaluées tant que l'une d'elles n'est pas *vrai*. Si aucune d'elles n'est *vrai*
  - les instructions associées au `else` sont exécutées
  - avant de poursuivre l'exécution des instructions suivantes (...).

Ordre des conditions

Un seul des blocs d'instructions est exécuté, celui qui correspond à la première condition évaluée à vrai. L'ordre dans lequel les différentes conditions sont énumérées est donc important.

Soit l'exemple suivant qui permet de savoir dans quelle catégorie est un ou une sportive en tir à l'arc :

```
if age <= 10:
    categorie = "poussin"
if 10 < age and age <= 12:
    categorie = "benjamin"
if 12 < age and age <= 14:
```

```
    categorie = "minime"
if 14 < age and age <= 17:
    categorie = "cadet"
if 17 < age and age <= 20:
    categorie = "junior"
else:
    categorie = "senior"
```

L'utilisation d'une construction `elif` avec un *ordre des conditions bien choisi* peut permettre de simplifier les conditions :

```
if age <= 10:
    categorie = "poussin"
elif age <= 12:
    categorie = "benjamin"
elif age <= 14:
    categorie = "minime"
elif age <= 17:
    categorie = "cadet"
elif age <= 20:
    categorie = "junior"
else:
    categorie = "senior"
```

Attention, un changement dans l'ordre de ces conditions ne permettra plus d'avoir la bonne catégorie en fonction de l'âge.

Memento

- la construction `if` … `elif` … `else` permet d'exécuter différents blocs d'instructions en fonction de différentes conditions
- les conditions d'une construction `if` … `elif` … `else` sont énumérées tant que l'une d'elles n'est pas vraie
- un seul des blocs d'instructions d'une construction `if` … `elif` … `else` est exécuté

Exercices niveau débutant

Premières fonctions

Minimum, maximum et valeur absolue

100. Définissez une fonction `maximum()` prenant en paramètre deux nombres et renvoyant le plus grand de ces deux nombres.
     On n'utilisera bien entendu pas la fonction Python prédéfinie `max()`.
101. Définissez une fonction `minimum()` prenant en paramètre deux nombres et renvoyant le plus petit de ces deux nombres.
     On n'utilisera bien entendu pas la fonction Python prédéfinie `min()`.
102. Définissez une fonction `valeur_absolue()` prenant en paramètre un nombre et renvoyant la valeur absolue de ce nombre.
     On n'utilisera bien entendu pas la fonction Python prédéfinie `abs()`.

Pair et signe

103. Définissez une fonction `parite()` prenant en paramètre un nombre entier et renvoyant
     - la chaîne de caractères `'pair'` si l'entier est pair
     - la chaîne de caractères `'impair'` si l'entier est impair
104. Définissez une fonction `signe()` prenant en paramètre un nombre et renvoyant une chaîne de caractères :
     - `'positif'` si le nombre est strictement positif
     - `'négatif'` si le nombre est strictement négatif
     - `'nul'` si le nombre est nul

Pile ou face

105. Définissez une fonction `pile_ou_face()` simulant le lancer d'une pièce de monnaie non biaisée, et renvoyant aléatoirement la chaîne de caractères `'pile'` ou la chaîne de caractères `'face'`.
     On utilisera une fonction du module `random` de Python pour simuler l'aléatoire.

Flots d'exécution

Au tir à l'arc, les catégories sont les suivantes :

- Poussins : 10 ans et moins
- Benjamins : 11 à 12 ans
- Minimes : 13 à 14 ans
- Cadets : 15 à 17 ans
- Juniors : 18 à 20 ans
- Seniors : 21 ans et plus

Ci-dessous différentes fonctions contiennent des instructions conditionnelles afin d'essayer de déterminer la catégorie d'un ou une sportive en fonction de son âge. Anticipez les valeurs renvoyées par ces fonctions dans les cas où le ou la sportive a 10 ans, a 15 ans, a 23 ans.

106. Quelles est ou quelles sont la ou les fonctions qui donne(nt) les résultats attendus ?
     Pourquoi les autres ne fonctionnent pas ?

a. Une suite de `if`

```
def categorie_tir_arc_v1(age):
    if age <= 10:
        categorie = "poussin"
    if age <= 12:
        categorie = "benjamin"
    if age <= 14:
        categorie = "minime"
    if age <= 17:
        categorie = "cadet"
    if age <= 20:
        categorie = "junior"
    else:
        categorie = "senior"
    return categorie
```

b. Des `elif` à la place des `if`

```
def categorie_tir_arc_v2(age):
    if age <= 10:
        categorie = "poussin"
    elif age <= 12:
        categorie = "benjamin"
    elif age <= 14:
        categorie = "minime"
    elif age <= 17:
        categorie = "cadet"
    elif age <= 20:
        categorie = "junior"
    else:
        categorie = "senior"
    return categorie
```

c. Changement dans l'ordre des `elif`

```
def categorie_tir_arc_v3(age):
    if age <= 10:
        categorie = "poussin"
    elif age <= 20:
        categorie = "junior"
    elif age <= 14:
        categorie = "minime"
    elif age <= 17:
        categorie = "cadet"
    elif age <= 12:
        categorie = "benjamin"
    else:
        categorie = "senior"
    return categorie
```

d. Changement des conditions et de l'ordre des `elif`

```
def categorie_tir_arc_v4(age):
    if age <= 10:
        categorie = "poussin"
    elif age <= 12:
        categorie = "benjamin"
    elif age <= 14:
        categorie = "minime"
    elif age > 20:
        categorie = "senior"
    elif age > 17:
        categorie = "junior"
```

```python
    else:
        categorie = "cadet"
    return categorie
```

Pour aller plus loin

Min, max, médiane à trois

107. Définissez deux fonctions `maximum3()` et `minimum3()` prenant en paramètre trois nombres et renvoyant respectivement le plus grand et le plus petit de ces trois nombres.
    On n'utilisera bien entendu pas les fonctions Python prédéfinies `max()` ou `min()`.
    1. Proposez éventuellement une version utilisant vos fonctions `maximum()` et `minimum()` définies précédemment.
    2. Proposez une version n'utilisant aucune autre fonction.
108. Définissez une fonction `mediane()` qui accepte trois nombres en paramètre et renvoie le nombre médian.

Exercices niveau intermédiaire

Saison

Définissez une fonction qui renvoie une des chaînes de caractères *'été'*, *'automne'*, *'hiver'*, *'printemps'* en fonction de la saison d'une date donnée par son numéro de jour et de mois. On supposera les débuts des saisons suivants :

- printemps le 20 mars
- été le 21 juin
- automne le 22 septembre
- hiver le 21 décembre

Dans l'ordre

109. Proposez une fonction `tri2` qui renvoie une chaîne de caractères présentant *dans l'ordre*, comme sur l'exemple suivant, deux nombres fournis en paramètre :

```
>>> tri2(1, 3)
'1 ≤ 3'
>>> tri2(3, 1)
'1 ≤ 3'
```

On pourra s'inspirer du code ci-dessous pour créer la chaîne de caractères :

```
>>> a = 1
>>> b = 3
>>> str(a) + " ≤ " + str(b)
'1 ≤ 3'
```

110. Proposez maintenant une fonction `tri3()` acceptant deux nombres en patamètres :

```
>>> tri3(8, 12, 7)
'7 ≤ 8 ≤ 12'
>>> tri3(8, 7, 12)
'7 ≤ 8 ≤ 12'
>>> tri3(12, 8, 7)
'7 ≤ 8 ≤ 12'
```

111. Combien de comparaisons sont effectuées pour trier les trois valeurs ?
Il est possible de proposer une méthode en trois comparaisons.

112. Proposez maintenant de trier quatre valeurs.

Retour sur le 421

Nous avions défini un prédicat pour déterminer si un lancer de trois dés forme un 421 à l'aide d'expressions booléennes – cf. « Expressions booléennes — niveau intermédiaire ».
On propose ici des fonctions alternatives pour ce même problème.

113. Proposez un prédicat à base d'instructions conditionnelles (et alternatives) imbriquées pour déterminer si trois dés forment un 421.

En supposant que les trois dés sont ordonnés, c'est-à-dire que la valeur du premier dé est supérieure ou égale à la valeur du second qui est elle-même supérieure à la valeur du troisième, permet de simplifier le test de savoir si les 3 dés forment un 421.

114. Proposez une version de `est_un_421()` suivant ce principe. La fonction commencera donc par trier les trois dés pour terminer par un test de la combinaison.

Impression à tarif dégressif

Une imprimerie applique le tarif suivant pour l'impression de flyers :

- 0,30€ l'unité pour 100 exemplaires
- 0,14€ l'unité pour 250 exemplaires
- 0,09€ l'unité pour 500 exemplaires
- 0,06€ l'unité pour 750 exemplaires
- 0,05€ l'unité pour 1000 exemplaires
- 0,03€ l'inité pour 2000 exemplaires
- 0,02€ l'unité à partir de 3000 exemplaires

Il n'est pas possible de commander d'autres quantités que celles listées, ou des multiples de 1000 au delà de 3000.

115. Proposez une fonction `nombre_exemplaires()` qui indique combien d'exemplaires doivent être commandés pour un nombre de flyers voulus.
Par exemple pour 400 flyers, il sera nécessaire d'en commander 500.

116. Proposez une fonction `montant_facture()` qui donne le prix à payer pour un nombre de flyers voulus.
Par exemple pour 400 flyers voulus, il est nécessaire d'en commander 500 pour un montant de 500 × 0,09€ = 45€.

Autour du calendrier

Nous complétons notre série de fonctions de manipulation de dates et de calendriers débutée avec le prédicat `est_bissextile()` et rappelé ci-dessous. Ce travail sera poursuivi lors de prochaines séances.

Année bissextile

Rappel de la feuille d'exercices « Expressions booléennes — niveau intermédiaire ».

L'année compte habituellement 365 jours. Les années bissextiles comptent 366 jours.

Depuis l'ajustement du calendrier grégorien, l'année est bissextile :

- si l'année est divisible par 4 et non divisible par 100, ou
- si l'année est divisible par 400.

Ainsi, 2019 n'était pas bissextile.
L'année 2020 est bissextile suivant la première règle : divisible par 4 et non divisible par 100.
L'an 1900 n'était pas bissextile car divisible par 4 mais aussi par 100, et non divisible par 400.
L'an 2000 était bissextile car divisible par 400.

(source Wikipédia `fr.wikipedia.org/wiki/Année_bissextile`).

117. Définissez un prédicat `est_bissextile()` :

```python
def est_bissextile(annee):
    """
    Paramètres :
    - annee : entier - l'année à tester
    Valeur de retour : booléen
    - True si annee est une année bissextile
    - False sinon
    Contraintes :
    - annee > 1582, année d'établissement du calendrier grégorien
    Exemples :
    >>> est_bissextile(2019)
    False
    >>> est_bissextile(2020)
    True
```

```
    >>> est_bissextile(1900)
    False
    >>> est_bissextile(2000)
    True
    """
```

Validité d'une date

Il est classique de représenter une date par trois nombres entiers (j, m, a), a désignant l'année, m le mois et j le numéro du jour dans le mois.

Mais bien entendu, tout triplet d'entiers ne représente pas nécessairement une date valide :

- $(4, -1, 2020)$, $(4, 15, 2020)$ ne sont pas des dates valides parce que les entiers -1 et 15 ne désignent pas des mois
- $(-2, 1, 2020)$, $(32, 1, 2020)$, $(29, 2, 2018)$ ne sont pas des dates valides parce que les entiers -2, 32 et 29 ne désignent pas des jours valides pour les mois correspondant

Nous allons définir deux prédicats préliminaires pour déterminer

- la validité d'un numéro de mois
- la validité d'un numéro de jour dans un mois donné d'une année donnée.

Validité d'un numéro de mois

118. Définissez un prédicat `est_mois_valide()` :

```
def est_mois_valide(m):
    """
    Test de validité d'un numéro de mois
    Paramètre :
    - m : entier
    Valeur de retour : booléen
    - True si m représente un mois valide, ie est compris entre 1 et 12
    - False sinon
    Contraintes : aucune
    Exemples :
    >>> est_mois_valide(-1)
    False
    >>> est_mois_valide(15)
    False
    >>> import random
    >>> est_mois_valide(random.randint(1, 12))
    True
    """
```

Validité d'un numéro de jour dans un mois

Déterminer la validité d'un numéro de jours dans un mois donné d'une année donnée nécessite de connaître le nombre de jours du mois concerné.

119. Définissez une fonction `nombre_jours()` :

```
def nombre_jours(mois, annee):
    """
    Nombre de jours d'un mois donné d'une année donnée
    Paramètres :
    - mois, annee : entiers - le mois et l'année
    Valeur de retour :
    - entier - le nombre de jours dans le mois mois de l'année annee.
    Contrainte :
    - mois est un mois valide
    Exemples :
    >>> nombre_jours(1, 2020)
    31
```

```
    >>> nombre_jours(2, 2018)
    28
    >>> nombre_jours(2, 2020)
    29
    """
```

120. À l'aide des fonctions précédentes, définissez maintenant une fonction `est_jour_valide()`

```
def est_jour_valide(j, mois, annee):
    """
    Validité d'un numéro de jour pour un mois et une année donnée
    Paramètres :
    - j : entier - le jour à tester
    - mois, annee : entier - les mois et année
    Valeur de retour : booléen
    - True si j est un jour valide pour le mois mois de l'année annee
    - False sinon
    Contraintes :
    - mois et annee valides
    Exemples :
    >>> est_jour_valide(-1, 1, 2020)
    False
    >>> est_jour_valide(32, 1, 2020)
    False
    >>> est_jour_valide(29, 2, 2018)
    False
    >>> est_jour_valide(29, 2, 2020)
    True
    """
```

Validité d'une date

121. Proposez un prédicat pour vérifier la validité d'une date donnée sous la forme de trois valeurs.
On vérifiera que
    - le mois est valide
    - le jour est valide
    - la date est postérieure au 15 octobre 1582, que l'on peut considérer comme premier jour du calendrier grégorien.

Nom du jour d'une date

Commençons par déterminer le nom d'un jour dont on connaît le numéro. Les numéros de noms de jour (à ne pas confondre avec les numéros de jours) sont des entiers entre 0 et 7.
Les valeurs 0 et 7 désignent dimanche, la valeur 1 désigne lundi, 2 mardi, etc. jusque 6 qui désigne samedi.

122. Définissez une fonction `nom_jour()` :

```
def nom_jour(njour):
    """
    Le nom d'un jour de numéro de jour donné
    Paramètre :
    - njour : entier - numéro d'un jour, 0 pour dimanche, 1 pour lundi, ...
      6 pour samedi, 7 pour dimanche
    Valeur de retour :
    - chaîne de caractères - le nom du jour
    Contraintes :
    - njour compris entre 0 et 7
    Exemples :
    >>> nom_jour(0)
    'dimanche'
    >>> nom_jour(1)
    'lundi'
    >>> nom_jour(6)
```

```
    'samedi'
    >>> nom_jour(7)
    'dimanche'
    """
```

Le numéro de nom de jour d'une date donnée peut être calculé selon différentes méthodes (cf Wikipédia ou Wikibooks). Une méthode est la suivante :

> Soit une date donnée sous la forme d'un triplet de nombres (j, m, a).
> Le nombre d calculé selon la formule
>
> $$d = (j + a - c + a_1 - ab + k + q) \bmod 7$$
>
> est un entier entre 0 et 6 compris, numéro du nom du jour.

Dans cette formule,

- $c = (14 - m) \div 12$ vaut 1 lorsque m vaut 1 ou 2, et 0 lorsque m vaut 3, 4, ..., 12
- ab est la partie *séculaire* de l'année a, autrement dit le nombre formé des deux premiers chiffres de l'année (par exemple si $a = 2017$, $ab = 20$)
- a_1 est le quotient entier de la division de a par 4 (par exemple si $a = 2017$, $a_1 = 504$)
- k est le quotient entier de la division de ab par 4 (par exemple pour $a = 2017$, $k = 5$)
- q est le quotient entier de la division de $31(m + 12c - 2)$ par 12 (par exemple pour $m = 9$, $q = 18$).

123. Définissez une fonction `num_jour()`

```
def num_jour(jour, mois, annee):
    """
    Numéro du jour dans la semaine d'une date donnée
    Paramètre :
    - jour : entier - le numéro du jour dans le mois
    - mois : entier - le numéro du mois dans l'année
    - annee : entier - l'année
    Valeur de retour :
    - un entier, numéro du jour dans la semaine, de 0 (dimanche)
      à 6 (samedi)
    Contrainte :
    - le triplet (jour, mois, annee) doit être une date valide
    Exemples :
    >>> num_jour(1, 1, 2019)
    2
    >>> num_jour(1, 1, 2100)
    5
    """
```

124. Utilisez les fonctions précédentes pour vérifier que le 28 septembre 2020 est bien un lundi, vérifier le nom de jour d'aujourd'hui, ou trouver un quel jour de la semaine vous êtes né·e.

Exercices niveau confirmé

Sans `or` ni `and`

On désire écrire des fonctions pour les opérations booléennes *and*,*or*, et *not*… sans utiliser les opérateurs Python **`and`**, **`or`**, et **`not`**, ni aucun autre opérateur Python.

On utilisera exclusivement les instructions conditionnelles.

125. Définissez une fonction `myown_not()` :

```python
def myown_not(a):
    """
    Équivalent de l'opérateur not
    Paramètre :
    - a : booléen
    Valeur de retour : booléen
    Contrainte : aucune
    Exemples (exhaustifs) :
    >>> myown_not(False)
    True
    >>> myown_not(True)
    False
    """
```

126. Définissez une fonction `myown_or()`. On se souciera particulièrement de respecter la propriété séquentielle de l'opérateur, c'est-à-dire de ne pas évaluer le second opérande quand cela n'est pas nécessaire.

```python
def myown_or(a, b):
    """
    Équivalent de l'opérateur or
    Paramètres :
    - a, b : booléens
    Valeur de retour : booléen
    Contrainte : aucune
    Exemples (exhaustifs) :
    >>> myown_or(False, False)
    False
    >>> myown_or(False, True)
    True
    >>> myown_or(True, False)
    True
    >>> myown_or(True, True)
    True
    """
```

127. Définissez de même une fonction `myown_and()` :

```python
def myown_and(a, b):
    """
    Équivalent de l'opérateur and
    Paramètres :
    - a, b : booléens
    Valeur de retour : booléen
```

```
    Contrainte : aucune
    Exemples (exhaustifs) :
    >>> myown_and(False, False)
    False
    >>> myown_and(False, True)
    False
    >>> myown_and(True, False)
    False
    >>> myown_and(True, True)
    True
    """
```

128. Si le second opérande des fonctions `myown_or()` ou `myown_and()` n'était pas de type booléen, quelle serait la valeur et le type renvoyés par vos fonctions quand l'exécution séquentielle ne peut être appliquée ?
129. Le comportement décrit à la question précédente est celui de Python.

     Donnez quelques expressions Python qui pourraient être utilisées pour mettre cela en évidence.

Sans `else` ni `elif`

Nous désirons (mais pourquoi donc ?) écrire des codes Python équivalents aux constructions conditionnelles `if` … `else` et `if` … `elif` …`else` sans utiliser le `else` ni le `elif`.

Équivalent du `else`

En première approximation, le code suivant

```
if condition_1:
    instructions_1
else:
    instructions_2
instruction_n
```

peut être remplacé par

```
if condition_1:
    instructions_1
if not condition_1:
    instructions_2
instruction_n
```

130. Montrez en exhibant un contre-exemple que les deux codes précédents ne sont pas équivalents.
131. Proposez deux fonctions `f_else()` et `f_else_contre_exemple()` construites sur la base de votre contre-exemple et correspondant aux schémas suivants.

```
def f_else(a):
    """
    Paramètre :
    - a : booléen
    Valeur de retour : booléen
    - vrai si et seulement si la partie if a été exécutée
    Exemples :
    >>> f_else(True)
    True
    >>> f_else(False)
    False
    """
    if condition_1 :
        instructions_1
        res = True
    else :
        instructions_2
        res = False
    return res
```

```python
def f_else_contre_exemple(a):
    """
    Paramètre :
    - a : booléen
    Valeur de retour : booléen
    (Contre-)Exemples :
    >>> f_else_contre_exemple(True) == f_else(True)
    True
    >>> f_else_contre_exemple(False) == f_else(False)
    True
    """
    if condition_1 :
        instructions_1
        res = True
    if not condition_1 :
        instructions_2
        res = False
    return res
```

Il s'agit de remplacer les `condition_1`, `instructions_1` et `instructions_2` par le code proposé dans votre contre-exemple.

Équivalent du `elif`

On considère maintenant un code tel que

```python
if condition_1:
    instructions_1
elif condition_2:
    instructions_2
elif condition_3:
    instructions_3
else:
    instructions_4
instruction_n
```

132. Proposez un code équivalent qui n'utilise ni `else` ni `elif`.

    Menez une démarche identique à celle de l'exercice précédent, « Équivalent du `else` ».

5. Interaction avec l’utilisateur

Interaction avec l'utilisateur — Programme

Coup d'œil

On découvre

- ce que sont les interactions avec l'utilisateur et les entrées/sorties
- comment écrire des programmes Python et les exécuter dans un terminal

Interaction avec l'utilisateur et entrées/sorties

Interaction avec l'utilisateur

L'*interaction avec l'utilisateur* permet à des programmes de gérer les échanges avec l'utilisateur, c'est-à-dire avec la personne qui utilise le programme, la personne qui utilise l'ordinateur.
Par exemple :

- un programme peut afficher un texte sur l'écran,
- un programme peut récupérer une entrée saisie au clavier,
- un programme peut dessiner sur l'écran, afficher une image, ou jouer une animation,
- un programme peut jouer un son, voire répondre à des commandes vocales,
- etc.

Entrées/sorties

Parmi l'ensemble de ces interactions, les *entrées/sorties* désignent particulièrement les échanges en mode écrit : entrées par le clavier et affichage de texte sur l'écran.

Pour réaliser ces entrées/sorties, des instructions sont proposées dans les langages de programmation.

En Python, on utilise les fonctions prédéfinies `print()` et `input()` que nous découvrirons bientôt.

Entrées/sorties via un terminal

En tant que *programmeur* Python, nous avons jusque maintenant travaillé dans un environnement de programmation – Thonny en ce qui nous concerne –.

En tant qu'*utilisateur*, nous allons exécuter les programmes Python dans un *terminal*.
Il s'agit d'une fenêtre texte dans laquelle il est possible d'exécuter des commandes du système d'exploitation, de lancer des programmes, dont les programmes que nous avons écrits en tant que programmeur Python !

Programme principal

Un programme est un peu plus qu'un ensemble de définitions de fonctions et d'instructions Python telles que nous les avons écrites jusque maintenant.

Il est en particulier nécessaire d'identifier les instructions à exécuter pour débuter l'exécution du programme.

Programme principal Python

Un programme, on dit aussi *programme principal* (*main* en anglais), est un ensemble de définitions de fonctions – parfois de variables – contenu dans un fichier. On parle aussi de *script*.

Une bonne pratique est de définir, plutôt en fin de ce fichier, une fonction particulière qui sera le point d'entrée de l'exécution du programme.
Cette fonction sera par exemple nommée `main()`, ou `mon_programme_main()`.

En toute fin de fichier, on précisera par les lignes suivantes qu'un appel à cette fonction doit être réalisé quand débute l'exécution du programme :

```
if __name__ == '__main__':
    main()
```

Un fichier `.py` correspondant à un programme possédera donc une structure comme

```
#
# Mon premier programme Python
#

# définition d'une fonction
def f():
    ...
    return ...
...

# définition de main, la fonction principale
def main():
    ...
    ... = f()
    ...

# code exécuté à l'exécution du programme
if __name__ == '__main__':
    main()
```

Annexe — Exécuter un programme dans un terminal depuis Thonny

Exécuter un programme dans un terminal nécessite donc

- de créer, par exemple dans l'environnement Thonny, un fichier `.py` correspondant au programme,
- de lancer un terminal pour y exécuter ce programme

On utilisera par exemple la commande « *Exécuter le script courant dans un terminal* » du menu « *Exécuter* » de Thonny (ou « *Run current script in terminal* » du menu « *Run* »).

Memento

- on distingue les rôles de *programmeur* et d'*utilisateur* d'un programme
- les entrées/sorties permettent à un programme d'interagir avec l'utilisateur
- cette interaction textuelle s'effectue lors de l'exécution d'un programme dans un terminal
- un programme principal comporte une fonction «*main*», point d'entrée de l'exécution du programme

Écrire sur le terminal

Coup d'œil

On écrit et exécute notre premier programme !
On découvre

- comment écrire du texte sur le terminal
- que les valeurs de tout type peuvent être écrites sur le terminal

Écrire sur le terminal

La fonction Python prédéfinie `print()` permet d'écrire sur le terminal.

Elle accepte un nombre variable de paramètres.

Les valeurs de chacun de ces paramètres sont écrites à la suite sur le terminal.

Hello world !

Notre premier programme va écrire un message de bonjour sur le terminal.
L'instruction Python pour écrire ce message est simplement

```
print('Hello world!')
```

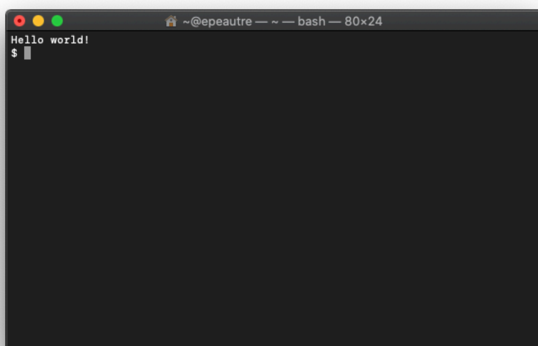
Le fichier Python qui contiendra notre programme sera donc

```
# Hello world !, mon premier programme Python

# point d'entrée de l'exécution de mon programme
def main():
    print('Hello world!')

# code exécuté à l'exécution du programme
if __name__ == '__main__':
    main()
```

Voici ce que donne une exécution dans le terminal :



Ce n'est qu'un premier programme !

(*Le caractère* `$` *affiché à la suite de l'exécution de notre programme est l'*invite *du terminal, on dit aussi* prompt. *Il est possible suite à cette invite de saisir d'autres commandes, de lancer l'exécution d'autres programmes.*)

La fonction `print()`

Écriture de tout type

La fonction `print()` écrit donc les valeurs de ses paramètres.

Ces paramètres peuvent être de n'importe quel type. Pour ceux que nous connaissons pour le moment :

- une chaîne de caractères – la suite de caractères est écrite, les caractères les uns à la suite des autres,
- un entier – ses chiffres sont écrits les uns à la suite des autres,
- un flottant – selon sa valeur, il est écrit avec la notation simple ou la notation scientifique (voir le support de cours « *Nombres à virgule flottante* »),
- un booléen – les caractères `True` ou `False` sont écrits.

Par exemple, les instructions suivantes :

```
from math import pi

print('Hello world!')
print(42)
print(pi)
print(pi**42)
print(0 == 0)
```

écriront dans le terminal :

```
Hello world!
42
3.141592653589793
7.590924172052281e+20
True
```

Fin de ligne et séparateurs

Les valeurs des paramètres de `print()` sont écrites les unes à la suite des autres, séparées par une espace.

Un retour à la ligne est écrit en fin.

Par exemple, les instructions

```
print("Le résultat du calcul est", 5.1 * 2)
print("(12 + 3) * 5 = ", (12 + 3) * 5)

prix = 1.99
poids = 2
print("Les poires sont à", prix, "euros le kilo donc",
      poids, "kg de poires valent", prix*poids, "euros.")
```

produira l'affichage suivant dans le terminal :

```
Le résultat du calcul est 10.2
(12 + 3) * 5 =  75
Les poires sont à 1.99 euros le kilo donc 2 kg de poires valent 3.98 euros.
```

Fonction et procédure

La fonction `print()` est particulière : elle ne renvoie pas de valeur, elle écrit sur le terminal.

Une telle fonction est appelée *procédure*.

Notre fonction `main()`, point d'entrée de notre programme est elle aussi une procédure.

Memento

- on écrit des valeurs de tout type sur un terminal avec la procédure prédéfinie `print()`
- `print()` écrit sur une ligne les valeurs de ses paramètres séparées par une espace
- une procédure est une fonction qui ne renvoie pas de valeur.

Lire depuis le terminal

Coup d'œil

On écrit et exécute notre deuxième programme !
On découvre

- comment lire un texte depuis le terminal
- que des données de tout type peuvent être lues depuis le terminal

Lire depuis le terminal

La fonction Python prédéfinie `input()` permet de lire des caractères saisis par l'utilisateur via le terminal.

Elle renvoie la chaîne de caractères lue.

Hello you !

Notre second programme va demander son prénom à l'utilisateur et lui répondre par un message de bienvenue.

L'instruction Python pour lire ce message est simplement

```
prenom = input('Quel est votre prénom ? ')
```

Son exécution affichera le message d'invitation et permettra à l'utilisateur de saisir son prénom que l'on récupérera dans la variable `prenom`.

Le fichier Python qui contiendra notre programme sera par exemple

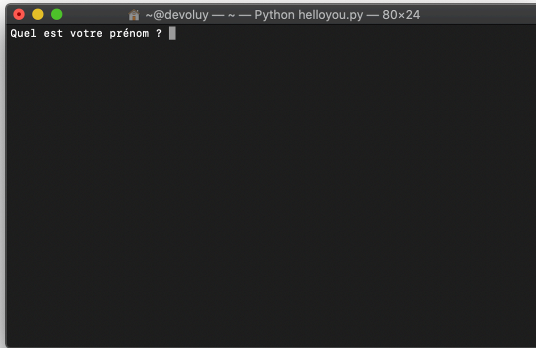
```
# Hello you !, mon deuxième programme Python

# point d'entrée de l'exécution de mon programme
def main():
    prenom = input('Quel est votre prénom ? ')
    print('Bienvenue', prenom, '!')

# code exécuté à l'exécution du programme
if __name__ == '__main__':
    main()
```

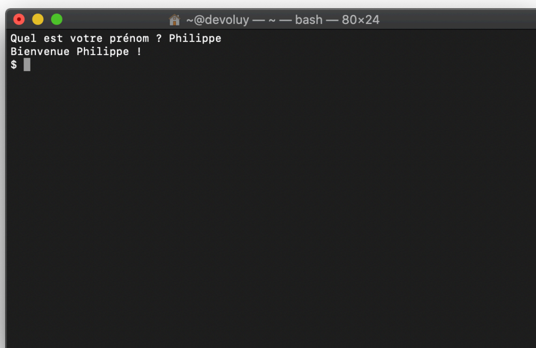
Voici les différentes étapes de l'exécution de ce programme dans le terminal :

1. Affichage de l'invitation



2. L'utilisateur saisi son prénom

3. Affichage du message de bienvenue



La fonction `input()`

Lecture de données de tout type

La fonction `input()` ne permet que de lire une chaîne de caractères.
Il est cependant possible de convertir la chaîne de caractères lue en une valeur d'un autre type (voir la section « *Conversions* » du support de cours « *Nombres à virgule flottante* »).

Par exemple le résultat d'une exécution dans le terminal des instructions suivantes

```
prenom = input('Quel est ton prénom ? ')
naissance = int(input('Quel ton année de naissance ? '))
print(prenom, 'tu as', 2020-naissance, 'ans ;)')
```

pourra être

```
Quel est ton prénom ? Raymond
Quel ton année de naissance ? 1978
Raymond tu as 42 ans ;)
```

Memento

- on lit des chaînes de caractères depuis le terminal avec la fonction prédéfinie `input()`
- les fonctions prédéfinies de conversions telles `int()` ou `float()` permettent de lire des données de tout type

Exercices niveau débutant

Premiers `print()` et `input()`

Fonction mystère

Voici le code d'une fonction mystère :

```
def mystere(n) :
    if n == 0 or n == 2 or n == 3 :
        print('oooo')
    elif n == 1 :
        print('   o')

    if 1 <= n and n <= 3 :
        print('   o')
    elif n == 0 :
        print('o  o')

    if n > 1 and n < 4 :
        print('oooo')
    elif n == 1 :
        print('   o')
    elif n == 0 :
        print('o  o')

    if n == 2 :
        print('o   ')
    elif n == 1 or n == 3:
        print('   o')
    elif n == 0 :
        print('o  o')
    else :
        print('...')

    if n == 0 or n == 2 or n == 3 :
        print('oooo')
    elif n == 1 :
        print('   o')
```

133. Écrivez les lignes affichées dans le terminal par l'exécution de chacune des 5 instructions suivantes :

```
mystere(0)
mystere(1)
mystere(2)
mystere(3)
mystere(4)
```

Saisie de données

134. Écrivez une suite d'instructions permettant de demander à l'utilisateur de saisir deux nombres entiers et affichant la somme de ces nombres, comme dans l'exemple ci-après

```
Donnez deux nombres entiers ? 12
15
La somme de 12 et 15 est 27
```

135. Écrivez un programme dont l'exécution dans un terminal permettra à l'utlisateur de saisir deux nombres entiers et affichant la somme de ces nombres.

Caractères spéciaux

Les caractères spéciaux, ne sont pas écrits tels quels par la fonction `print()`, mais permettent de passer à la ligne ou d'écrire des caractères non accessibles directement.

On les note sous la forme du caractère `\` – antislash ou barre oblique inverse – suivi d'un autre caractère, par exemple `\n`.
Cette barre oblique inverse n'est pas comprise comme un caractère, il ne sera pas écrit comme tel. De même pour le caractère qui suit la barre oblique.

Parmi les caractères spéciaux, notons :

- `\n` qui permet d'insérer un passage à la ligne sur le terminal
- `\t` qui permet d'insérer une tabulation
- `\"` et `\'` qui permettent d'écrire les guillemets
- `\\` qui permet d'écrire le `\` lui-même.

136. Anticipez le résultat de l'exécution des instructions suivantes

```
str1 = "Écrivons une première ligne\n et une seconde ligne\n"
str2 = 'Écrivons \n\t - des guillemets simples \' et doubles \"\n'
str3 = "\t - des barres obliques inversées \\ ou pas /"
print(str1 + str2 + str3)
```

Utilisez Thonny pour visualiser ce que produit cette exécution dans un terminal.

Ces caractères spéciaux sont écrits avec deux symboles, mais forment bien un unique caractère :

137. Évaluez les expressions suivantes :

```
>>> len('\n')
>>> len("\\")
```

Jeu du '421'

L'objectif est de réaliser une version très simplifiée du '421' consistant en un seul lancer de 3 dés 6 mais permettant une interaction avec l'utilisateur via le terminal.
Vous pourrez réutiliser des fonctions déjà écrites précédemment.

Commencez par créer un répertoire `mini_jeux` dans lequel vous sauvegarderez un script `quatre_deux_un.py` que vous allez compléter.

À la fin de ce script, définissez une fonction principale `main_421()` qui sera le point d'entrée du programme.
Les fonctions ci-après permettent de décomposer ce programme en fonctions plus simples, indépendantes les unes des autres et qui seront toutes utilisées dans cette fonction principale.

Fonctions propres au jeu du '421'

138. Définissez une fonction `de()` sans paramètre simulant le lancer d'un dé à 6 faces.

139. Définissez un prédicat `est_un_421` acceptant 3 paramètres correspondant aux valeurs de chacun des dés et testant si ces valeurs forment un triplet 421 (dans n'importe quel ordre).

Interaction avec l'utilisateur

Il s'agit maintenant d'ajouter quelques fonctions permettant :

- d'afficher des informations à destination de l'utilisateur
- de lui demander de fournir certaines informations.

140. Définissez une fonction `identification()` sans paramètre qui demandera à l'utilisateur de s'identifier à l'aide d'un nom et qui **renverra** cette valeur sous forme d'une chaîne de caractères
141. Définissez une procédure `presentation()` acceptant en paramètre le nom de l'utilisateur. Elle **affichera** un message de bienvenue personnalisé et expliquera très brièvement le principe du jeu.
142. Définissez un prédicat `poursuivre_le_jeu()` qui a pour objectif de demander à l'utilisateur s'il souhaite tenter sa chance au jeu en lançant les dés. L'utilisateur répondra par exemple par « oui » ou « non », « yes » ou « no ». Ce prédicat **renverra** la valeur `True` si et seulement si le joueur désire poursuivre.

Affichage sur le terminal

La mise en forme des résultats va permettre au jeu d'être plus convivial.

143. Définissez une procédure `affiche_lancer()` acceptant trois paramètres correspondant aux valeurs de chacun des dés.
Cette fonction réalisera un affichage mis en forme des dés comme celui ci-dessous. Respectez strictement les caractères utilisés dans l'exemple :

```
>>> affiche_lancer(4, 3, 5)

+---+   +---+   +---+
| 4 |   | 3 |   | 5 |
+---+   +---+   +---+
```

144. Définissez enfin une procédure `affiche_resultat()` acceptant les trois mêmes paramètres et affichant un message de victoire ou d'échec suivant la réussite (peu probable en un essai !) du lancer.

One main to rule them all

Il vous reste à compléter la fonction principale point d'entrée de l'exécution du programme.
Elle appellera les fonctions que vous venez de définir dans un certain ordre.

145. Définissez une procédure `main_421()` sans paramètre.
Cette fonction devra, dans l'ordre :
    - procéder à l'identification de l'utilisateur
    - présenter le jeu
    - vérifier que le joueur souhaite jouer
    - si c'est bien le cas :
        - lancer chacun des dés
        - afficher le lancer
        - afficher le résultat du jeu (réussite ou non)
    - sinon : afficher un message adapté d'au revoir.
146. On sait comment lancer automatiquement l'ensemble des doctests lors de l'exécution du programme :

```
if __name__ == '__main__' :
    import doctest
    doctest.testmod(verbose = True)
```

On peut ajouter à ce bloc d'instruction conditionnelle l'appel à la fonction `main_421()`. Que va-t-il alors se passer lors de l'exécution de votre programme dans un terminal ?

147. Lancer l'exécution de votre programme dans un terminal.

Exercices niveau intermédiaire

Jeu du bandit manchot

Différents modèles de machine à sous, aussi appelés *bandits manchots*, existent dans les établissements de jeu. Un modèle simple consiste en trois rouleaux comportant cinq symboles, représentés ici par les caractères `$%#~&`. Les combinaisons gagnantes et les gains correspondants sont indiqués ci-dessous :

Combinaison	Gain
trois `$`	mise × 250
trois `%`	mise × 150
deux `$`	mise × 5
un seul `$`	mise × 2
tout autre combinaison	0

On souhaite réaliser un jeu consistant en une seule utilisation de cette machine à sous : le joueur pourra miser une somme de son choix.
Dans le répertoire `mini_jeux` précédent, créez un script `bandit_manchot.py`.

Fonctions propres au jeu

148. Définissez une fonction `rouleau()` simulant la rotation d'un seul rouleau de la machine et renvoyant un caractère parmi ceux possibles.
149. Définissez une fonction `calcul_gain()` acceptant quatre paramètres :

    - la mise de départ du joueur
    - les 3 symboles correspondant à chaque rouleau

    Elle renverra le gain en suivant les indications fournies en amont.
150. Définissez une fonction `chaine_differentiel()` acceptant deux paramètres : la mise et le gain.
    Cette fonction renverra une chaîne de caractères présentant la différence entre la mise et le gain précédée d'un signe `+` ou `-` selon que le joueur a gagné ou perdu de l'argent.

    Par exemple, pour une mise de 100€ et un gain de 500€ :

```
>>> chaine_differentiel(100, 500)
'+400'
```

Interaction avec l'utilisateur

151. En prenant comme modèle le jeu du '421' définissez les fonctions équivalentes suivantes :

    - une fonction `identification()`
    - une fonction `presentation()`

Affichage sur le terminal

152. Définissez une fonction `affiche_rouleaux()` acceptant trois paramètres correspondant aux symboles des trois rouleaux après utilisation du bandit_manchot.
    L'affichage de ces rouleaux devra correspondre exactement à l'exemple ci-après. Il sera précédé d'une

phrase de présentation.
Pour un appel `affiche_rouleaux('%', '$', '&')`, le texte suivant sera écrit sur le terminal :

```
les symboles tirés au sort sont :
 ----- ----- -----
|  %  |  $  |  &  |
 ----- ----- -----
```

153. Définissez une fonction `affiche_gain()` acceptant un paramètre : le gain.
Cette fonction réalisera l'affichage de ce gain mis en forme. Les exemples `affiche_gain(150)` et `affiche_gain(0)` produiront respectivement

```
Bravo, vous avez gagné 150 €  :)
```

et

```
Aucun gain, pas de chance :(
```

154. Définissez enfin une fonction `affiche_bilan()` acceptant deux paramètres : la mise et le gain. Elle procédera à l'affichage du différentiel mis en forme, par exemple pour un appel `affiche_bilan(100, 500)` :

```
Par rapport à votre mise initiale, le différentiel est de : +400 €
```

Fonction principale

155. Définissez une fonction `main_bandit_machot()` sans paramètre.
Inspirez vous de la fonction principale du jeu du 421 mais n'oubliez pas de :
    - demander une mise de départ au joueur
    - d'afficher le bilan de la partie.
156. Ajoutez les instructions nécessaires pour qu'à l'exécution de ce script les doctests soient vérifiés et que la fonction principale soit aussi automatiquement lancée.
157. Exécutez votre programme dans un terminal.

Lanceur de mini-jeux

Notre collection de mini-jeux s'étend : il serait judicieux de réaliser un script permettant de lancer les autres scripts déjà codés. C'est ce que nous allons faire.
Toujours dans le répertoire `mini_jeux` créez un nouveau script nommé `mini_jeux.py`.

Voici un premier code proposé pour ce lanceur :

```python
from bandit_manchot import main_bandit_manchot

def main_games() :
    print('Jouons ensemble\n')
    jeu = input('À quel jeu voulez-vous jouer ?  B  (B si "bandit_manchot"')
    if (jeu == 'B') :
        main_bandit_manchot()

if __name__ =='__main__':
    main_games()
```

158. Expliquez le rôle de la première ligne.
Quelles sont les fonctions disponibles (vous pouvez vous aider du panneau « *Variables* » de Thonny).
159. Complétez le code ci-dessus pour permettre de jouer également au 421.
160. Exécutez votre programme `mini_jeux.py` dans un terminal.

Jeu de la roulette

Continuons notre collection de mini-jeux.
La roulette est un jeu de hasard dans lequel chaque joueur mise sur un ou plusieurs numéros, une couleur, la

hauteur ou la parité du numéro qu'il espère être tiré.
Le tirage du numéro s'effectue à l'aide d'une bille jetée dans un récipient circulaire tournant et muni d'encoches ayant des numéros de différentes couleurs (source : Wikipédia).
La roulette possède 37 cases numérotées de 0 à 36 alternativement rouges et noires, à l'exception du zéro qui est vert.
Pour notre jeu simplifié, à chaque tour de jeu, une des mises suivantes est possible :

- un nombre entier quelconque compris entre 1 et 36
- *Manque* pour les numéros de 1 à 18 (on "manque" la moitié)
- *Passe* pour les numéros de 19 à 36 (on "passe" la moitié)
- *Pair* pour un nombre pair
- *Impair* sur un nombre impair

161. Créez un script `roulette.py` dans le répertoire `mini-jeux`.

Prédicats

162. Définissez un prédicat `est_manque()` qui pour un numéro donné en paramètre, renvoie `True` si il est *Manque*, `False` sinon (il est alors *Passe*) :

```
>>> est_manque(2)
True
>>> est_manque(35)
False
```

163. Définissez un prédicat `est_impair()` qui pour un numéro donné en paramètre, renvoie `True` si il est est *Impair*, `False` sinon (il est alors *Pair*) :

```
>>> est_impair(2)
False
>>> est_impair(35)
True
```

Roulez jeunesse

164. Définissez une fonction `lancer()` qui simule un lancer de roulette et renvoie un numéro compris entre 0 et 36.

Interaction avec l'utilisateur

165. Définissez une fonction `demande_montant()` qui demande au joueur de saisir le montant qu'il veut miser et renvoie un entier qui représente ce montant.
Une interaction dans le terminal pourra être :

```
Quel est le montant de votre mise ? 20
```

166. Réalisez une fonction `demande_mise()` qui demande au joueur la mise qu'il désire faire renvoie la chaîne de caractères saisie par l'utilisateur.
Cette chaîne de caractères peut avoir les valeurs suivantes

    - un numéro compris entre '1' et '36'
    - 'Manque'
    - 'Passe'
    - 'Pair'
    - 'Impair'

    On considère que l'utilisateur ne saisit pas une autre chaîne. Une interaction dans le terminal pourra être :

```
Quelle mise voulez-vous faire ? Pair
```

ou

```
Quelle mise voulez-vous faire ? 1
```

Calcul des gains

Le gain est calculé en fonction de la mise du joueur et du numéro du lancer.
Les gains ne sont pas cumulables et sont les suivants :

- 36 fois le montant si le numéro misé est celui de la roulette
- 2 fois le montant le type joué (*Pair*, *Impair*, *Manque* ou *Passe*) correspond à celui du numéro de la roulette
- 1,5 fois le montant (arrondi au supérieur) si le numéro joué et celui de la roulette sont tous les deux *Impair* ou tous les deux *Pair*
- 1,5 fois le montant (arrondi à l'inférieur) si le numéro joué et celui de la roulette sont tous les deux *Manque* ou tous les deux *Passe*
- 0 dans tous les autres cas (la mise est perdue)

167. Écrire la fonction `calcul_gain()` qui prend en paramètres le numero du lancer de roulette, la mise choisie par le joueur et le montant misé, et qui renvoie le gain.

    Astuce : si la chaîne de caractères représentant la case fait moins de 3 caractères, le joueur a joué un nombre, si elle en fait plus, il a joué *Pair*, *Impair*, *Manque* ou *Passe*.

    ```
    >>> calcul_gain(6, "Pair", 21)
    42
    >>> calcul_gain(36, "Passe", 21)
    42
    >>> calcul_gain(6, "Impair", 21)
    0
    >>> calcul_gain(5, "5", 20)
    720
    >>> calcul_gain(5, "6", 21)
    31
    >>> calcul_gain(5, "31", 21)
    32
    ```

    Conseil : il pourra être avantageux de définir deux fonctions annexes par exemple `calcul_gain_numero()` et `calcul_gain_pari()` qui permettront de décomposer cette fonction.

Affichage, bilans, fonction principale...

168. Ajoutez les fonctions suivantes comme pour le script du bandit manchot : `presentation()`, `identification()`, `affiche_gain()`, `chaine_differentiel()`, `affiche_bilan()`

169. Définissez une fonction `affiche_lancer()` qui prend en paramètre le numéro tiré au sort et affiche la valeur de la case ainsi que sa parité et le caractère Passe ou Manque de cette case. Par exemple, pour les valeurs 23, 32, et 0, s'affichera respectivement :

    `23 Impair et Passe`

    `32 Pair et Passe`

    et

    `0 Tout le monde perd`

    Vous réutiliserez bien entendu les prédicats précédents.

170. Enfin, définissez la fonction principale `main_roulette()` qui s'inspirera très fortement de celle du bandit manchot. N'oubliez pas cependant de :

    - saisir la mise
    - d'afficher le résultat du lancer

171. Exécutez votre programme dans un terminal.

6. Boucles - for

Itérable et boucle ‘for‘

Coup d’œil

On découvre

- comment répéter l’exécution de certaines instructions sur une série de valeurs
- la notion d’itérable, structure qu’il est possible de parcourir
- comment parcourir une séquence de valeurs telle une chaîne de caractères

Dans certaines situations, on peut vouloir exécuter des mêmes instructions sur une série de valeurs – tels des caractères ou des nombres –. Il s’agit par exemple d’afficher ces valeurs, ou vérifier que des caractères sont des lettres, ou vérifier que des entiers sont pairs, etc.

Il nous faut pouvoir identifier :

- la série de valeurs : ce sera un *itérable*, notion que nous allons découvrir
- les instructions à exécuter : nous utiliserons la boucle `for` de Python

Itérable

Un *itérable* est une *séquence de valeurs* que l’on va pouvoir parcourir.

Une ou plusieurs instructions pourront être exécutées successivement sur chacune des valeurs. L’exécution de cette ou ces instructions sur une valeur est appelée une *itération*.

Les *chaînes de caractères* que nous connaissons sont des itérables. Il va être possible de répéter des instructions sur chacun des caractères de la chaîne.

Nous découvrirons d’autres itérables par la suite.

La boucle d’itération `for`

La boucle `for` permet la répétition d’un bloc d’instructions sur chacune des valeurs d’un itérable.

On écrira par exemple :

```
for c in 'aEi42' :
    print('*')
```

qui affichera une ligne avec une `*` pour chaque caractère de la chaîne *'aEi42'*.

Notez les mots clés

- `for` qui introduit la boucle, et
- `in` qui sépare un identificateur, ici `c`, et l’itérable.

Notez aussi le caractère `:` qui introduit le bloc d’instructions qui sera répété.

On pourra aussi écrire :

```
for c in 'aEi42' :
    print('*', c)
```

Notez l'utilisation de l'identificateur associé au `for` – ici le `c` –. Il s'agit d'une variable, dite *variable d'itération*, qui prend successivement la valeur de chacun des éléments de l'itérable – chacun des caractères de notre chaîne *'aEi42'*.

L'exécution de cette boucle `for` affiche

```
* a
* E
* i
* 4
* 2
```

dans le terminal.

Cet autre exemple illustre l'utilisation de l'indentation pour identifier le bloc d'instructions du corps de la boucle, et la fin de ce bloc.

(*islower() et isdigit() testent respectivement si le caractère est une minuscule ou un chiffre.*)

L'exécution de

```
for cc in 'aEi42' :
    print('----------------------')
    print(cc, '\t', cc.islower(), '\t', cc.isdigit())
print('======================')
```

écrit

```
----------------------
a    True    False
----------------------
E    False   False
----------------------
i    True    False
----------------------
4    False   True
----------------------
2    False   True
======================
```

sur le terminal.

Ce dernier exemple n'affiche que les lettres minuscules, à raison d'une par ligne :

```
for cc in 'aEi42' :
    if cc.islower() :
        print(cc)
print('--')
```

produit l'affichage de

```
a
i
--
```

Le nombre d'itérations (répétitions) du bloc d'instructions d'une boucle `for` est connu. Il correspond à la taille de l'itérable. Dans le dernier exemple, même si seules les lettres minuscules sont affichées, toutes les lettres de la chaîne *'aEi42'* sont testées, 5 répétitions sont effectuées.

Memento

- un *itérable* est une séquence de valeurs qui va pouvoir être parcourue à l'aide d'une boucle `for`
- les chaînes de caractères sont des itérables
- le bloc d'instructions d'une boucle `for` est répété pour chacune des valeurs de la séquence
- dans ce bloc d'instructions, la *variable d'itération* est associée à la valeur de l'itération courante

Intervalle ‘range‘

Coup d’œil

On découvre

- la notion d’intervalle d’entiers et comment construire de tels intervalles en Python
- comment parcourir des intervalles avec la boucle `for`

Intervalle

Un *intervalle* est une séquence d’entiers. Par exemple, la suite des entiers entre 2 et 6 forme un intervalle.

Un intervalle est un itérable. Il sera donc possible de parcourir la séquence des entiers avec une boucle `for`.

La construction Python `range()` permet de créer des intervalles. On utilisera `range(a, b)` pour créer l’intervalle des entiers successifs compris entre a (inclus) et b (exclu).

Par exemple `range(2, 6)` désignera la séquence d’entiers 2, 3, 4, 5.

D’autres utilisations de `range()` sont possibles que nous découvrirons en exercice.

Itérer sur un intervalle avec `for`

La boucle `for` peut être utilisée avec une construction `range()` pour exécuter des instructions sur chacune des valeurs d’un intervalle d’entiers.

On écrira par exemple

```
for i in range(2, 6) :
    print('*')
```

pour afficher quatre lignes avec une `*` dans le terminal.

On écrira par exemple

```
for i in range(2, 6) :
    print('*', i)
```

pour afficher chacun des entiers compris entre 2 et 6, 6 exclu :

```
* 2
* 3
* 4
* 5
```

dans le terminal.

L’exemple suivant illustre à nouveau l’utilisation de l’indentation pour identifier le corps de la boucle.

(*`i%2 == 0` est vrai pour les seuls entiers pairs.*)

L’exécution de

```
for i in range (2, 6) :
    print('-------')
```

```
    print(i, i%2 == 0)
print('=======')
```

écrit donc

```
-------
2 True
-------
3 False
-------
4 True
-------
5 False
=======
```

Et enfin, ce dernier exemple n'affiche que les entiers pairs de l'intervalle 2, 6 :

```
for i in range (2, 6) :
    if i%2 == 0 :
        print(i)
print('--')
```

soit :

```
2
4
--
```

Memento

- la construction `range()` de Python crée des intervalles d'entiers
- `range(a, b)` est la suite des $b - a$ entiers compris entre a et b, b exclu
- ces intervalles d'entiers sont des itérables
- une boucle `for` peut donc être utilisée pour parcourir un intervalle d'entiers

Utilisations classiques de boucles ‘for‘

Coup d’œil

On découvre

- les situations typiques d’utilisation de boucles `for`

Affichage

Une boucle `for` peut être utilisée pour afficher une à une les valeurs d’un itérable.

Les instructions de la boucle utiliseront la fonction `print()` pour afficher la valeur courante de la variable d’itération.

Les supports de cours précédents « *Itérable et boucle* `for` » et « *Intervalle* `range` » ont présenté quelques exemples sur les chaînes de caractères et les intervalles d’entiers.

L’ensemble des valeurs peut être affiché, ainsi que certaines propriétés de ces valeurs. Par exemple, nous avons vu comment :

- afficher le fait qu’un caractère est une lettre minuscule ou un chiffre, ou pas
- afficher le fait qu’un entier est pair ou pas

Il est possible de n’afficher que certaines des valeurs en combinant boucle `for` et instruction conditionnelle `if`. Par exemple, nous avons vu comment :

- n’afficher que les lettres minuscules
- n’afficher que les entiers pairs

Recherche, identification

En parcourant un itérable, il est possible de tester si une valeur donnée est ou non présente dans celui-ci.

Le principe est le suivant :

- une variable booléenne témoin est utilisée pour mémoriser le fait que l’on ait observé ou non cette valeur
- avant de parcourir l’itérable, nous n’avons pas encore observé la valeur recherchée. La variable booléenne témoin est donc initialisée à `False`
- lors du parcours, si la valeur recherchée est observée, la variable booléenne témoin passe à `True`

À la suite de la boucle, la variable témoin indique si la valeur recherchée a été observée, ou non.

Le prédicat ci-dessous recherche la présence de la lettre z (en majuscule ou minuscule) dans la chaîne passée en paramètre.

```
def contient_z(s):
    """
    :param s: (str)
    :contrainte: aucune
    :return: (bool) True si s contient 'z' ou 'Z'
    :exemples:
    >>> contient_z('Test')
    False
```

```
    >>> contient_z("Zoro")
    True
    >>> contient_z("Est-ce un zèbre ?")
    True
    >>> contient_z('')
    False
    """
    trouve = False
    for c in s:
        if c == 'z' or c == 'Z':
            trouve = True
    return trouve
```

Il est également possible de rechercher si un itérable contient au moins une valeur qui répond à un critère donné.

Le principe est le même.

Par exemple, le prédicat ci-dessous vérifie si

l'intervalle passé en paramètre contient un multiple de 11.

```
def contient_mult11(intervalle):
    """
    :param intervalle: (range)
    :contrainte: aucune
    :return: (bool) True si intervalle contient au moins un multiple de 11
    :exemples:
    >>> contient_mult11(range(1,5))
    False
    >>> contient_mult11(range(1,11))
    False
    >>> contient_mult11(range(10,30))
    True
    """
    trouve = False
    for i in intervalle:
        if i%11 == 0:
            trouve = True
    return trouve
```

Remarquons que l'exécution de la boucle `for` *se poursuit jusqu'à la fin de l'itérable, même quand la valeur recherchée est trouvée. Nous verrons dans le cours sur les boucles* `while` *qu'il est possible d'interrompre une boucle dès que la valeur recherchée est trouvée.*

Comptage, accumulation

Le parcours d'un itérable permet de compter le nombre de valeurs de l'itérable , ou – plus intéressant – compter le nombre de valeurs qui correspondent à un critère donné.

Le principe est

- d'initialiser une variable compteur à zéro
- dans la boucle `for`, d'incrémenter ce compteur à chaque fois que l'on observe une valeur qui répond au critère

À la fin de la boucle, le compteur indique le nombre de valeurs trouvées.

Un premier exemple pour compter le nombre de valeurs d'un itérable. (Notez que la la fonction prédéfinie `len()` renvoie cette valeur.)

```
def nvaleurs(iter) :
    """Nombre de valeur d'un itérable
    Paramètre : iter - itérable
    Contraintes : aucune
    Valeur de retour : entier - nombre de valeurs de l'itérable
    Exemples :
```

```
    >>> nvaleurs('aEi42')
    5
    >>> nvaleurs(range(2, 6))
    4
    """
    compteur = 0
    for i in iter :
        compteur = compteur + 1
    return compteur
```

La fonction suivante permet de compter le nombre de lettres minuscules d'une chaîne de caractères. On reconnaît l'usage de l'instruction conditionnelle `if` pour identifier les valeurs à comptabiliser.

```
def nminuscules(s) :
    """Nombre de lettres minuscules dans une chaîne de caractères
    Paramètre : s - chaîne de caractères
    Contraintes : aucune
    Valeur de retour : entier - nombre de lettres minuscules
    Exemples :
    >>> nminuscules('aEi42')
    2
    >>> nminuscules('aeiouy')
    6
    >>> nminuscules('')
    0
    """
    compteur = 0
    for c in s :
        if c.islower() :
            compteur = compteur + 1
    return compteur
```

Il est aussi possible d'accumuler les valeurs d'un itérable vérifiant un certain critère, par exemple concaténer certains caractères.

Le principe est identique au comptage. Initialement, le résultat est par exemple une chaîne vide. À chaque fois que l'on rencontre un caractère qui correspond au critère, on l'ajoute (c.-à-d. le concatène) au résultat.

La fonction suivante produit une chaîne de caractères formée des seules lettres minuscules d'une chaîne fournie en paramètre.

```
def minuscules(s) :
    """Chaîne de caractères formée des seules lettres minuscules d'une chaîne donnée
    Paramètre : s - chaîne de caractères
    Contraintes : aucune
    Valeur de retour : chaîne de caractères
    Exemples :
    >>> minuscules('aEi42')
    'ai'
    >>> minuscules('aeiouy')
    'aeiouy'
    >>> minuscules('MAX')
    ''
    """
    les_minuscules = ''
    for c in s :
        if c.islower() :
            les_minuscules = les_minuscules + c
    return les_minuscules
```

À suivre...

D'autres utilisation classiques des boucles `for` seront présentées plus tard. En particulier quand il nous sera possible de créer des « listes » ou des « ensembles » de valeurs de tout type.

Memento

- une boucle `for` peut être utilisée pour afficher les valeurs, ou certaines valeurs, d'un itérable
- une boucle `for` peut être utilisée pour vérifier la présence de certaines valeurs dans un itérable
- une boucle `for` peut être utilisée pour compter les valeurs, ou certaines valeurs, d'un itérable

Boucles ‘for‘ imbriquées

Coup d’œil

On découvre

- ce qu’est l’imbrication de boucles `for`, et quels en sont les usages
- comment produire les couples et les paires de valeurs prises dans plusieurs itérables

L’imbrication de boucles `for` consiste à écrire une instruction de boucle `for` dans le bloc d’instructions d’une autre boucle `for`. La première boucle est appelée boucle externe et la deuxième la boucle interne ou imbriquée.

L’imbrication de boucles `for` est utilisée pour parcourir plusieurs itérables « simultanément ». En fait, pour chaque itération de la boucle externe, la boucle interne est parcourue entièrement.

Ensemble des couples

L’imbrication de boucles est un moyen simple de réaliser un « produit cartésien » entre deux (éventuellement plus) séquences de valeurs.
Il s’agit de produire tous les couples de valeurs.

L’exemple suivant affiche les couples de valeurs dont la première composante est une des lettres de la chaîne `'ABC'` et la seconde composante un entier de l’intervalle $[1-3[$:

```
for c in 'ABC':
    for i in range(1, 3):
        print(c, i)
```

l’exécution produit le résultat suivant dans le terminal :

```
A 1
A 2
B 1
B 2
C 1
C 2
```

Ensemble des paires

Pour deux valeurs différentes données, le couple (a, b) est distinct du couple (b, a), l’ordre des valeurs est importante.
Il n’en est pas de même pour les paires : la paire $\{a, b\}$ est égale à la paire $\{b, a\}$, l’ordre des valeurs n’a pas d’importance.

L’imbrication de deux boucles `for` permet de produire l’ensemble des paires d’un itérable.

Par exemple, pour l’intervalle $[1, 2, 3, 4]$, l’ensemble des paires de valeurs distinctes est formé de $\{1, 2\}$; $\{1, 3\}$; $\{1, 4\}$ et $\{2, 3\}$; $\{2, 4\}$, et $\{3, 4\}$.

On peut utiliser le principe suivant pour produire un tel ensemble de paires de valeurs distinctes :

- pour la première valeur de l’itérable, il s’agit de produire les paires en combinant cette première valeur à chacune des *valeurs suivantes* de l’itérable

- pour la seconde valeur de l'itérable, il s'agit de produire les paires en combinant cette seconde valeur à chacune des *valeurs suivantes* de l'itérable
- …
- pour l'avant dernière valeur, il s'agit de produire la paire en combinant cette avant-dernière valeur avec la dernière valeur de l'itérable
- pour la dernière valeur de l'itérable… il n'y a pas de paire à produire.

Pour la i-ème valeur, il s'agit donc de produire les paires $\{i, j\}$ avec j appartenant à $]i, max[$.

La borne de début de la boucle imbriquée dépend donc de la valeur de la variable d'itération de la boucle externe.

L'exemple suivant affiche toutes les paires composées d'entiers distincts compris entre 1 et 4 inclus :

```
for i in range(1, 4):
    for j in range(i+1 ,5):
        print(i, j)
```

```
1 2
1 3
1 4
2 3
2 4
3 4
```

Memento

- on utilise des boucles `for` imbriquées pour produire l'ensemble des couples de deux itérables
- on utilise des boucles `for` imbriquées pour produire l'ensemble des paires d'un itérable
  - les bornes de la boucle interne dépendent alors de la valeur de la variable d'itération de la boucle externe

Exercices niveau débutant

Intervalles d'entiers : `range()`

Pour ces premiers exercices, on pourra se passer de rédiger une docstring complète.

172. Écrivez une fonction qui affiche dans la console la séquence des nombres compris entre 10 et 20 inclus.
173. Écrivez une fonction qui affiche dans la console la séquence des nombres compris entre 0 et 20 inclus.
174. Exécutez la suite d'instructions suivante

```
for i in range(6):
    print(i)
```

Qu'obtenez-vous ? Que représente le paramètre du `range()` lorsqu'il n'y en a qu'un ?

175. Exécutez ces deux nouvelles suites d'instructions

```
for i in range(10, 21, 2):
    print(i)
```

```
for i in range(105, 90, -1):
    print(i)
```

Qu'obtenez-vous pour chaque suite ? Que représente chacun des 3 paramètres du `range()` ?

176. Vous venez de découvrir que `range()` accepte de 1 à 3 paramètres. Les deux paramètres manquants sont facultatifs car ils ont des valeurs par défaut.

Quels sont les paramètres facultatifs et quelles sont leurs valeurs par défaut ?

177. Écrivez une expression `range()` avec trois paramètres qui soit équivalente à `range(10)`.

Recherche

Minuscules

178. Proposez un prédicat `sont_minuscules()` qui vérifie que tous les caractères d'une chaîne de caractères sont des lettres minuscules ou des espaces. On utilisera la fonction Python prédéfinie `islower()` pour vérifier qu'un caractère est une lettre minuscule :

```
>>> 'k'.islower()
True
>>> 'K'.islower()
False
>>> ' '.islower()
False
>>> sont_minuscules('des kakis et des kiwis')
True
>>> sont_minuscules('des kakis et des kiwis !')
False
>>> sont_minuscules('des Kakis et des Kiwis')
False
>>> sont_minuscules('')
True
```

Comptage, accumulation

Sommes

179. Écrivez une fonction `somme_entiers()` qui prend en paramètre un entier positif n et qui renvoie la somme des entiers de 1 à n :

$$1 + 2 + 3 + \cdots + (n-1) + n = \sum_{i=1}^{n} i$$

```
>>> somme_entiers(3) == 6
True
>>> somme_entiers(10) == 55
True
```

(*On produira cette somme par une boucle* **`for`**, *sans utiliser la « formule du petit Gauss ».*)

180. Écrivez une fonction `somme_carres()` qui prend en paramètre un entier positif n et qui renvoie la somme des carrés des entiers de 1 à n :

$$1^2 + 2^2 + 3^2 + \cdots + (n-1)^2 + n^2 = \sum_{i=1}^{n} i^2$$

```
>>> somme_carres(3) == 14
True
>>> somme_carres(10) == 385
True
```

181. Écrivez une fonction `somme_impairs()` qui prend en paramètre un entier positif n et qui renvoie la somme des n premiers entiers impairs :

$$1 + 3 + 5 + \cdots + (2n-1) = \sum_{i=1}^{n} (2i-1)$$

```
>>> somme_impairs(3) == 9
True
>>> somme_impairs(10) == 100
True
```

182. Écrivez une fonction `somme_alternee()` qui prend en paramètre un entier positif n et qui renvoie

$$-1 + \frac{1}{2} - \frac{1}{3} + \ldots + \frac{(-1)^n}{n} = \sum_{i=1}^{n} \frac{(-1)^i}{i}$$

On pourra éviter de calculer directement $(-1)^i$ en remarquant que cette valeur « alterne » pour les valeurs de i successives.

```
>>> epsilon = 10e-10
>>> somme_alternee(10) - somme_alternee(9) <= 1/10 + epsilon
True
>>> 1/10 - epsilon < somme_alternee(10) - somme_alternee(9)
True
```

Produits

183. Écrivez une fonction `factorielle()` qui accepte un paramètre entier positif n et qui renvoie la valeur $n!$ (factorielle n) définie comme le produit des entiers de 1 à n :

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n = \prod_{i=1}^{n} i$$

```
>>> factorielle(5) == 120
True
>>> factorielle(1) == 1
True
```

184. Écrivez une fonction `puissance()` qui prend en paramètre deux entiers n et p et qui renvoie n élevé à la puissance p, soit n^p, en utilisant uniquement l'opérateur de multiplication.

```
>>> puissance(3, 0) == 1
True
>>> puissance(4, 1) == 4
True
>>> puissance(1, 5) == 1
True
>>> puissance(2, 3) == 8
True
```

Comptages

185. Écrivez une fonction qui compte le nombre de caractères d'une chaîne, sans utiliser la fonction Python `len()` :

```
def nombre_caracteres(s):
    """
    Renvoie le nombre de caractères d'une chaîne donnée, sans utiliser
    la fonction prédéfinie len().
    Parametre : s - chaîne de caractères
    Valeur de retour : entier - le nombre de caractères de s
    Contrainte : aucune
    Exemples :
    >>> nombre_caracteres("Test") == 4
    True
    >>> nombre_caracteres("Un autre test. ") == 15
    True
    >>> nombre_caracteres('') == 0
    True
    """
```

186. Écrivez une fonction qui compte le nombre d'espaces présents dans une chaîne de caractères :

```
def nombre_espaces(s):
    """
    Renvoie le nombre d'espaces présents dans une chaîne donnée
    Paramètre : s - chaîne de cractères
    Valeur de retour : entier - le nombre d'espaces
    Contrainte : aucune
    Exemples :
    >>> nombre_espaces("Test") == 0
    True
    >>> nombre_espaces("Un autre test. ") == 3
    True
    >>> nombre_espaces('') == 0
    True
    """
```

187. Écrivez une fonction qui compte le nombre de signes de ponctuation suivants : ".", "?", "!" dans une chaîne :

```
def nombre_ponctuations(s):
    """
    Renvoie le nombre de caractères  ".", "?", "!" présents dans une chaîne donnée
    Paramètre : s - chaîne de cractères
    Valeur de retour : entier - le nombre de caractères ".", "?", "!"
    Contrainte : aucune
    Exemples :
    >>> nombre_ponctuations("Test.") == 1
    True
    >>> nombre_ponctuations("Un autre test ! Pour voir.") == 2
    True
```

```python
    >>> nombre_ponctuations('') == 0
    True
    """
```

188. Écrivez une fonction qui compte le nombre d'occurrences d'un caractère donné dans une chaîne de caractères :

```python
def occurrences_caractere(s, carac):
    """
    Renvoie le nombre d'occurence d'un caractère donné dans une chaîne
    Paramètres :
    - s : chaîne de caractères
    - carac : chaîne de (un) caractère
    Valeur de retour : entier - le nombre d'occurrences de carac dans s
    Contrainte : len(carac) == 1
    Exemples :
    >>> occurrences_caractere("Test.", 't') == 1
    True
    >>> occurrences_caractere("Un autre test ! Pour voir.", 't') == 3
    True
    >>> occurrences_caractere('', 't') == 0
    True
    """
```

Exercices niveau intermédiaire

Accumulation

Suite

189. Écrivez une fonction `terme_suite()` qui prend en paramètre un entier positif n et qui renvoie la valeur du terme u_n de la suite définie par

$$\begin{cases} u_0 = 1 \\ \forall\, n \geq 1, \quad u_n = 2u_{n-1} + 3 \end{cases}$$

Exemples :

```
>>> terme_suite(0) == 1
True
>>> terme_suite(1) == 5
True
>>> terme_suite(2) == 13
True
```

(*On procédera avec une boucle* **`for`**, *sans utiliser la formule de calcul du terme général d'une suite arithmético-géométrique.*)

Table de multiplications

190. Définissez une fonction `chaine_multiplication()` paramétrée par 2 entiers a et b et qui renvoie la **chaîne** de la forme « a x $b = c$ », où c est le résultat de la multiplication, comme dans l'exemple ci-dessous.

```
>>> chaine_multiplication(6, 7) == '6 x 7 = 42'
True
```

191. Définissez une fonction `table_multiplications()` paramétrée par un entier n et renvoyant une chaîne contenant la table de multiplications de n entre 0 et 10, présentée comme ci-dessous.

```
>>> table_multiplications(3) == ' | 0 x 3 = 0 | 1 x 3 = 3 | 2 x 3 = 6 | 3 x 3 = 9 | 4 x 3 = 12 | 5
True
```

Boucles imbriquées

Chaîne des paires

Nous souhaitons produire une chaîne de caractères pour écrire toutes les paires d'un itérable donné.

192. Proposez une fonction préliminaire `paire2str()` :

```
def paire2str(a, b):
    """Renvoie une écriture de la paire {a, b}
    Paramètres :
    - a, b : entiers - composantes de la paire
    Valeur de retour : str - l'écriture de la paire
    Contraintes : aucune
    Exemples :
    >>> paire2str(2, 2)
```

```
    '{2,2}'
    >>> paire2str(2, 6)
    '{2,6}'
    """
```

193. Proposez une fonction `les_paires()` qui renvoie une chaîne de caractères contenant l'ensemble des paires d'entiers compris entre 1 et une valeur n donnée en paramètre. Cet ensemble de paires inclut les paires dont les deux composantes sont égales. Par exemple :

```
>>> les_paires(2)
'{1,1} {1,2} {2,2} '
>>> les_paires(4)
'{1,1} {1,2} {1,3} {1,4} {2,2} {2,3} {2,4} {3,3} {3,4} {4,4} '
```

Une seconde version pourra éviter de produire un caractère espace en fin de chaîne :

```
>>> les_paires(2)
'{1,1} {1,2} {2,2}'
>>> les_paires(4)
'{1,1} {1,2} {1,3} {1,4} {2,2} {2,3} {2,4} {3,3} {3,4} {4,4}'
```

Affichage de motifs

194. Écrivez une procédure `affiche_ligne()` qui prend en paramètre un entier n positif et qui affiche à l'écran une séquence de caractères `0` de taille n. Cette procédure ne doit pas utiliser l'opérateur `*` et s'écrit à l'aide d'une seule boucle, voir l'exemple ci-dessous.

```
>>> affiche_ligne(5)
00000
>>> affiche_ligne(3)
000
```

195. Écrivez une première version d'une procédure `affiche_bloc()` qui prend en paramètre un entier n positif et qui affiche à l'écran un bloc de n lignes composées de n caractères `0`, comme dans l'exemple ci-dessous. Cette version doit utiliser la procédure `affiche_ligne()`.

```
>>> affiche_bloc(3)
000
000
000
```

196. Écrivez une deuxième version d'une procédure `affiche_bloc()` qui utilise deux boucles imbriquées.

197. De même, écrivez deux versions d'une procédure `affiche_triangle()` qui produit l'affichage suivant :

```
>>> affiche_triangle(3)
0
00
000
```

198. Écrivez une fonction `affiche_diagonale()` qui met un `X` sur la diagonale du bloc et un `0` dans le reste du bloc, voir l'exemple ci-dessous.

```
>>> affiche_diagonale(3)
X00
0X0
00X
```

Exercices niveau confirmé

Version provisoire qui sera complétée

Accumulation

Majuscules

199. Proposez une fonction `en_majuscules()` qui transforme les caractères d'une chaîne de caractères qui sont des lettres minuscules en majuscule. Les autres caractères restent inchangés. On utilisera la fonction Python prédéfinie `upper()` qui renvoie la lettre majuscule correspondant à une lettre minuscule :

```
>>> 'k'.upper()
K
>>> en_majuscules('des kakis et des kiwis')
'DES KAKIS ET DES KIWIS'
>>> en_majuscules('des kakis et des kiwis !')
'DES KAKIS ET DES KIWIS !'
>>> en_majuscules('des Kakis et des Kiwis')
'DES KAKIS ET DES KIWIS'
>>> en_majuscules('')
''
```

200. Proposez une fonction `en_capitale()` qui remplace dans une chaîne passée en paramètre

- la première lettre, et
- chacune des lettres précédées d'une espace

par la lettre majuscule correspondante :

```
>>> en_capitale('des kakis et des kiwis')
'Des Kakis Et Des Kiwis'
>>> en_capitale('Des Kakis Et Des Kiwis')
'Des Kakis Et Des Kiwis'
>>> en_capitale('DES KAKIS ET DES KIWIS')
'DES KAKIS ET DES KIWIS'
>>> en_capitale('')
''
```

Chaîne miroir

201. Proposez une fonction `miroir()` qui produit le miroir d'une chaîne de caractères :

```
>>> miroir('Python')
'nohtyP'
>>> miroir('super')
'repus'
>>> miroir('')
''
```

Anacyclique et palindrome

Un *anacyclique* est un mot ou un groupe de mots qui conserve un sens lorsqu'on le lit de droite à gauche

Par exemple, « zen » et « nez » ou « super » et « repus » sont anacycliques. C'est aussi le cas de « l'ami naturel » et « le rut animal » si l'on ne tient pas compte des caractères qui ne sont pas des lettres.

Dans le cas général, le texte est différent selon que l'on lit dans le sens normal de lecture ou en sens inverse. Lorsque le texte est identique, il s'agit d'un *palindrome*.

Par exemple, « été » et « kayak » sont des palindromes. C'est aussi le cas de la phrase « Engage le jeu que je le gagne. ».

(*source : page Wikipédia « Anacyclique » fr.wikipedia.org/wiki/Anacyclique. Consultez cette page pour découvrir d'autres anacycliques.*)

On dira que deux mots ou groupes de mots sont *frères anacycliques* s'ils forment un couple d'anacycliques, lecture l'un de l'autre de droite à gauche.

L'objectif de l'exercice est de déterminer si deux chaînes de caractères données sont frères anacycliques. Donc de proposer un prédicat `anacycliques()` :

```
>>> anacycliques('zen', 'nez')
True
>>> anacycliques("l'ami naturel", 'le rut animal')
True
```

Une première version pourra être « stricte » et tenir compte de tous les caractères, sans distinguer lettres et autres caractères.

Une version standard pourra procéder par étape : supprimer l'ensemble des caractères qui ne sont pas des lettres avant de réaliser la comparaison des chaînes.

Une version avancée pourra ne pas tenir compte des signes diacritiques (accents, trémas, cédilles) et ainsi confondre « e », « é », et « ê », ou « c » et « ç ».

Annexe

On pourra utiliser la primitive `isalpha()` pour déterminer si un caractère est une lettre :

```
>>> 'e'.isalpha()
True
>>> 'E'.isalpha()
True
>>> '3'.isalpha()
False
>>> '!'.isalpha()
False
>>> 'é'.isalpha()
True
```

Exercices complémentaires

Une fiche d'exercices indépendants de la progression habituelle débutant / intermédiaire / confirmé.

Au menu :

1. découverte de la tortue et du module `turtle` de Python
2. dessiner des carrés, dessiner avec des carrés
3. dessiner avec des ordres (L-Systèmes)

Dessiner avec le module `turtle`

La *tortue* désigne un robot virtuel capable de se déplacer dans un plan en laissant une trace ou non de son passage. Elle a été inventée dans les années 1960 au MIT par Seymour Papert qui a conçu un langage informatique – Logo – et une philosophie de l'éducation (voir la page Wikipédia « Logo (langage) » fr.wikipedia.org/wiki/Logo_(langage)).

De nombreux langages informatiques ont repris cette idée de tortue. Le module `turtle` met en œuvre une telle tortue.

On utilise les fonctions du module après un `import turtle`.

Il est possible de commander le paramétrage du crayon par :

- `turtle.down()` qui abaisse le stylo
- `turtle.up()` qui relève le stylo
- `turtle.pencolor(color)` qui change la couleur (`'red'`, `'green'`, `'blue'`, etc.)

On déplace la tortue avec :

- `turtle.forward(length)` qui avance d'un nombre de pas donné;
- `turtle.backward(length)` qui recule;
- `turtle.right(angle)` qui tourne vers la droite d'un angle donné (en degrés);
- `turtle.left(angle)` qui tourne vers la gauche.

On peut également déplacer la tortue à un point donné ou modifier son orientation avec :

- `turtle.goto(x, y)` qui déplace la tortue jusqu'au point (x, y);
- `turtle.setheading(angle)` qui oriente la tortue à l'angle donné en degrés, le 0° étant à l'est, le 90° au nord, etc.

La fenêtre par défaut est 950 pixels en largeur et 800 pixels en hauteur. Le point $(0, 0)$ est au centre de l'écran. Au départ, la tortue est en $(0, 0)$, orientée à 0°.

D'autres fonctionnalités sont disponibles, on consultera éventuellement la documentation `docs.python.org/fr/3/library/t`

La majorité des primitives fournies par le module `turtle` sont des procédures. Elles ne renvoient pas de valeur. Elles modifient l'état du « système », le « système » étant compris comme la fenêtre dans laquelle évolue la tortue, et l'état étant compris comme ce qui est dessiné sur cette fenêtre, la position de la tortue, son orientation, etc.

Voici un exemple:

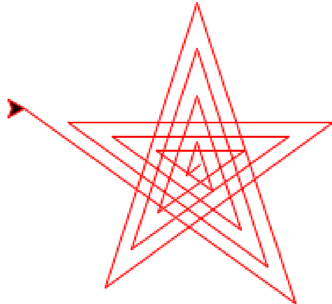
```
import turtle

def spiral(n) :
    turtle.pencolor("red")
```

```
    for i in range(n):
        turtle.forward(i * 10)
        turtle.right(144)
```

Essayez :

```
>>> spiral(20)
```

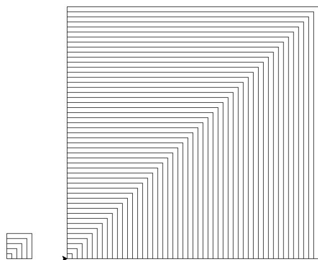


Dessiner avec des carrés

202. Proposez une procédure `carre()` qui dessine un carré dont la longueur des côtés est fournie en paramètre. Ce carré sera dessiné depuis l'état courant de la tortue. À la fin du dessin l'état de la tortue sera identique à son état initial.

203. Proposez une procédure `cinq_carres()` qui dessine cinq carrés emboîtés ayant comme sommet commun la position initiale de la tortue. Les longueurs des carrés sont comprises entre 10 à 50 (inclus). Les longueurs de chacun des carrés varient de 10 en 10.

     Vous pouvez également proposer une variante `cinquante_carres()` qui dessine cinquante carrés de longueurs comprises entre 10 et 500 inclus. C'est plus joli, mais il est plus difficile de vérifier qu'il y a bien 50 carrés dessinés.

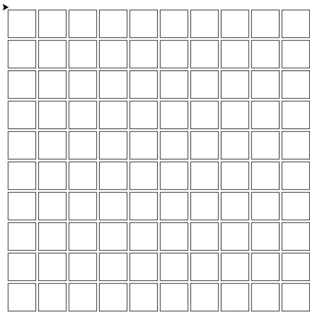
     

204. Proposez une procédure `dix_carres()` qui dessine dix carrés de côtés 50 espacés de 5 comme sur la figure.

     Vous pouvez proposer une variante qui accepte en paramètre la longueur des côtés des carrés.

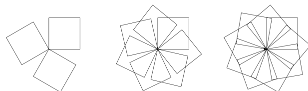
     Remarquez qu'à l'issue de l'exécution de la procédure, la tortue se trouve à la base du dernier carré dessiné.

205. Proposez une procédure `cent_carres()` qui dessine dix lignes de dix carrés comme sur la figure.

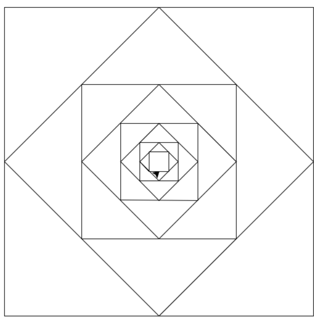


206. Proposez une procédure `carres_tournants()` qui dessine n carrés de côté 100 pivotant autour d'une sommet commun pour faire un tour complet. L'angle entre deux carrés successifs est donc de $360 \div n$ degrés. Cette procédure est paramétrée par le nombre de carrés à dessiner.

    Voici le résultat pour 3, 7, et 9 carrés.

207. Proposez une procédure `carres_emboites()` qui dessine de manière répétée un certain nombre de carrés emboités comme dans la figure ci-dessous (exemple pour 9 carrés). Cette procédure admet un paramètre, la longueur du carré le plus grand. Le dessin peut se faire en dessinant les carrés depuis le plus grand jusqu'au plus petit. La taille de deux carrés successifs est alors dans un rapport de $\sqrt{(2)}$.

Dessiner avec des ordres

On souhaite donner une *séquence* d'ordres à la tortue sous forme d'une chaîne de caractères.

Étant donnés

- une longueur l
- un angle α exprimé exprimé en degrés

La tortue interprètera chacun des caractères de la chaîne comme une commande à effectuer :

caractère	ordre
`'F'`	avancer de l
`'G'`	avancer de l (identique à `'F'`)
`'+'`	tourne vers la gauche de α
`'-'`	tourne vers la droite de α

Par exemple, la chaîne de caractères `'++F-F--F-F'` pour des valeurs de $l = 100$ et $\alpha = 45$ permettrait de tracer une « maison ».

Dessiner un ordre

208. Proposez une procédure `dessine()` qui prend en paramètres :

    - une chaîne de caractères, séquence d'ordres

- une longueur,
- un angle,

et réalise à l'aide la tortue le tracé correspondant aux ordres.

Testez cette procédure sur l'exemple précédent.

209. Proposez des procédures utilisant un unique appel à `dessine()` et réalisant le tracé

    - d'un carré
    - d'un hexagone régulier

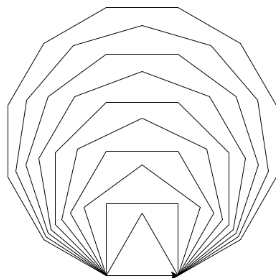
    Chacune de ces procédures accepte un paramètre correspondant à la longueur de la figure à tracer.

210. Proposez une procédure généralisant les précédentes pour dessiner un polygone régulier. Cette procédure acceptera un paramètre entier indiquant le nombre de côtés, et un paramètre correspondant à la longueur de la figure à tracer.

211. Proposez une procédure qui trace l'ensemble des polygones réguliers entre deux nombres de côtés *min* et *max* donnés en paramètre. Par exemple

    ```
    >>> dessine_des_polygones(3, 12, 100):
    ```

    produira le dessin suivant

Dériver un ordre

Dériver un ordre consiste à remplacer chacun des caractères `'F'` d'un ordre donné par une autre chaîne de caractères.

Par exemple, la dérivation de `'F+F'` par `'F-F'` produit `'F-F+F-F'`.

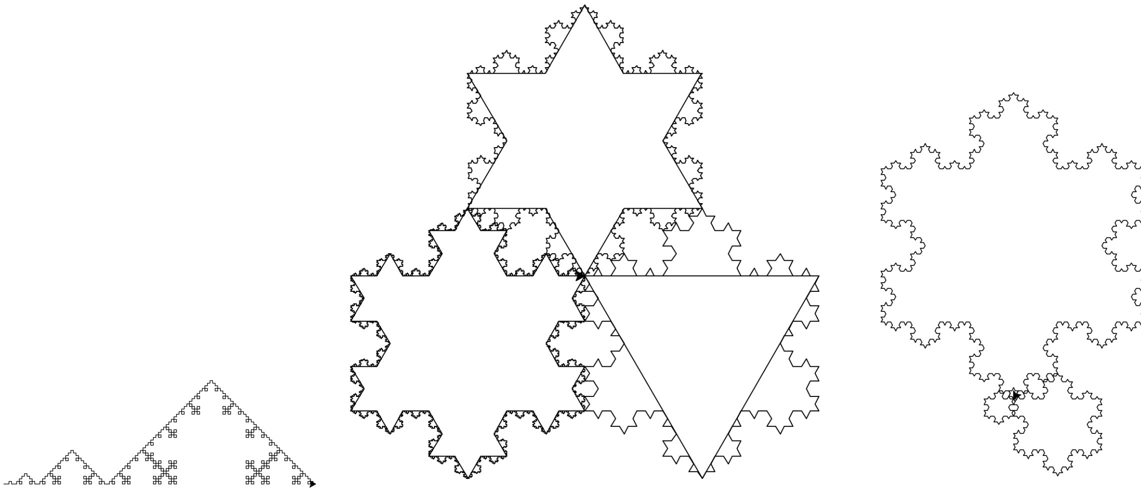
212. Proposez une fonction `derive()` qui accepte deux chaînes en paramètres, et renvoie la dérivation de la première par la seconde.

Il est possible de dériver plusieurs fois de suite un ordre par une autre chaîne de caractères. On obtient alors la dérivation n-ième de l'ordre par la chaîne.

Par exemple les dérivations successives de `'F'` par `'+F-'` produisent :

```
>>> derive_nieme('F', '+FF', 1)
'+FF'
>>> derive_nieme('F', '+FF', 2)
'++FF+FF'
>>> derive_nieme('F', '+FF', 3)
'+++FF+FF++FF+FF'
```

213. Proposez une fonction `derive_nieme()` qui accepte deux chaînes et un entier n en paramètres, et renvoie la dérivation n-ième de la première chaîne par la seconde.



Pour dessiner les figures suivantes, utilisez la fonction `speed()` du module `turtle` pour accélerer le déplacement de la tortue.

214. Proposez une fonction `koch()` qui accepte un paramètre entier n et dessine les dérivations n-ième de `'F'` par `'F+F-F-F+F'` pour n variant de 1 à 4, et pour une longueur de 4 et un angle de 90°

215. Proposez une fonction `flocons()` qui accepte un paramètre entier n et dessine les dérivations n-ième de `'F--F--F` par `'F+F--F+F'` pour une longueur de 40 et un angle de 60°.

216. Proposez une fonction pour tracer les dérivations n-ième, pour n compris entre 0 et 5, une longueur $l = 3^{5-n}$ et un angle $\alpha = 60$ de la séquence d'ordres `'F--F--F'` par la chaîne `'F+F--F+F'`.

Consultez la page Wikipédia « L-Système » fr.wikipedia.org/wiki/L-Système pour découvrir d'autres *fractales* à dessiner.

7. Chaînes de caractères

Rappels sur les chaînes de caractères

Coup d'œil

On révise
- ce qu'est une chaîne de caractères
- les opérateurs applicables aux chaînes
- les fonctions et méthodes liées aux chaînes

Chaînes de caractères

Les chaînes de caractères sont des séquences de caractères et sont donc des itérables. Les caractères sont des symboles qui peuvent être écrits, que ce soit des lettres, des chiffres, des symboles de ponctuation ou des lettres d'autres alphabets.

La chaîne vide ne contient aucun caractère.

Chaîne de caractères se dit *string* en anglais, et le type Python associé est `str`

Itération sur les chaînes

La boucle `for` permet la répétition d'un bloc d'instructions sur chaque caractère d'une chaîne.

Le code ci-dessous compte le nombre de lettres minuscules d'une chaîne *s* à l'aide de la variable *compteur*.

```
compteur = 0
for c in s :
    if c.islower() :
        compteur = compteur + 1
```

Caractères spéciaux

Une chaîne de caractères littérale s'écrit entre délimiteurs, `'` ou `"`.
Certains caractères ne peuvent pas être écrits tels quels, par exemple les délimiteurs eux-mêmes.
On utilise alors le **caractère d'échappement** \ (contre-barre ou anti-slash) avant le caractère spécial.

La table suivante présente quelques exemples de caractères spéciaux :

écriture	caractère	exemple de chaîne	écriture Python
`\"`	guillemet	Des "guillemets".	`"Des \"guillemets\"."`
`\'`	apostrophe	Des 'apostrophes'.	`'Des \'apostrophes\'.'`
`\n`	passage à la ligne	Ligne 1 Ligne 2	`"Ligne 1\nLigne 2"`
`\t`	tabulation	Avant Après	`"Avant\tAprès"`
`\\`	contre-barre	2 contre-barres \ et \	`"2 contre-barres \\ et \\"`

Remarquez que l'écriture Python donnée en dernière colonne n'est qu'une des écritures possibles (et pas toujours la plus simple).

Opérateurs applicables aux chaînes

Concaténation

L'opérateur + concatène deux chaînes : il produit une nouvelle chaîne composée des deux chaînes mises bout à bout.

```
>>> s1 = "Je commence. "
>>> s2 = "Tu finis."
>>> s1 + s2
'Je commence. Tu finis.'
```

Répétition

L'opérateur $*$ appliqué à un entier n et une chaîne répète la chaîne n fois.

```
>>> s1 = "Ah ! "
>>> s1 * 3
'Ah ! Ah ! Ah ! '
```

Égalité entre chaînes

Deux chaînes de caractères sont égales si et seulement si elles contiennent exactement les mêmes caractères, dans le même ordre.
Les lettres minuscules et majuscules sont distinctes.

```
>>> 'test' == 'Test'
False
>>> "C'est cool !" == 'C\'est cool !'
True
>>> 'Ah ! Ah ! Ah ! ' != 3 * "Ah ! "
False
```

Fonctions et méthodes liées aux chaînes

Le tableau ci-dessous reprend les fonctions et méthodes prédéfinies liées aux chaînes de caractères déjà rencontrées :

nom	exemple d'expression	valeur de l'expression
`len()`	`len("Test")`	`4`
`str()`	`str(3.1416)`	`'3.1416'`
`int()`	`int("12")`	`12`
`float()`	`float('3.1416')`	`3.1416`
`str.islower()`	`"Test".islower()`	`False`
	`"test".islower()`	`True`
`str.isupper()`	`"Test".isupper()`	`False`
	`"TEST".isupper()`	`True`
`str.isdigit()`	`"Test".isdigit()`	`False`
	`"12".isdigit()`	`True`

Pour plus d'informations sur ces fonctions et pour en découvrir d'autres, vous pouvez consulter la documentation du langage Python : docs.python.org/fr/3/library/stdtypes.html?#string-methods.

Memento

- une chaîne de caractères est une séquence de caractères et donc un itérable
- les opérateurs + et $*$ sont utilisés pour concaténer ou répéter des chaînes de caractères
- les opérateurs == et != sont utilisés pour comparer les chaînes
- Python propose de nombreuses fonctions et méthodes prédéfinies pour manipuler les chaînes

Indices pour les chaînes de caractères

Coup d'œil

On découvre
- comment accéder à un caractère donné d'une chaîne
- une nouvelle façon de parcourir une chaîne à l'aide de boucles `for`
- comment accéder à une sous-chaîne

Accès à un caractère d'une chaîne

Une chaîne est une séquence ordonnée de caractères.
Chacun des caractères peut être désigné par sa place dans la séquence à l'aide d'un **indice**.
On parle aussi d'index.

L'accès à un caractère se réalise simplement en faisant suivre la chaîne par l'indice du caractère entre crochets. Par exemple :

```
>>> s = "Informatique"
>>> s[2]
'f'
```

Remarquez que l'indice 2 ne correspond pas au second mais au troisième caractère.
En effet, les caractères d'une chaîne sont indicés **à partir de zéro** (et non à partir de un).

L'indice du dernier caractère de la chaîne est donc égal à la longueur de la chaîne moins un. Par exemple avec la chaîne `s` définie précédemment :

```
>>> s[len(s) - 1]
'e'
```

Attention, toute tentative d'accès à un caractère en dehors de la chaîne, se traduit par une `IndexError`, erreur d'indice, indice en dehors de l'intervalle :

```
>>> s[12]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
IndexError: string index out of range
```

Itération sur les indices

Les indices des caractères d'une chaîne sont donc compris entre 0 inclus, et la longueur de la chaîne, exclue.

Une boucle `for` parcourant cet intervalle va permettre d'accéder successivement aux caractères de la chaîne. Par exemple :

```
s = "Test"
for i in range(len(s)):
    print("----------------")
    print(i, "\t", s[i])
print("================")
```

L'affichage obtenu est le suivant :

```
-----------------
0	T
-----------------
1	e
-----------------
2	s
-----------------
3	t
=================
```

L'itération sur les indices permet de parcourir différemment les chaînes, par exemple en allant à l'envers :

```
s = "Test"
for i in range(len(s)-1, -1, -1):
    print("-----------------")
    print(i, "\t", s[i])
print("=================")
```

qui affiche

```
-----------------
3	t
-----------------
2	s
-----------------
1	e
-----------------
0	T
=================
```

ou uniquement les caractères d'indice pair :

```
s = "Test"
for i in range(0, len(s), 2):
    print("-----------------")
    print(i, "\t", s[i])
print("=================")
```

qui affiche

```
-----------------
0	T
-----------------
2	s
=================
```

Autres indiçages

L'accès à un caractère d'une chaîne par indiçage est une opération courante dans la plus part des langages de programmation.
Python propose d'autres indiçages.

Accès à une sous-chaîne

Une **tranche** d'une chaîne de caractères est une séquence de caractères consécutifs de la chaîne.

Les indices d'une tranche forment un intervalle : de l'indice du premier caractère (inclus) à l'indice du dernier caractère (exclu).

L'accès à une tranche se fait en écrivant cet intervalle entre crochets, comme par exemple :

```
>>> s = "Informatique"
>>> s[0:4]
'Info'
```

La valeur d'une telle expression est une chaîne de caractères, sous-chaîne de la chaîne accédée.

Accès avec un indice négatif

Python autorise l'indiçage des caractères d'une chaîne à parti de la fin de chaîne.
Ces indiçages se font avec des valeurs d'indice négatives.

L'indice -1 est utilisé pour désigner le dernier caractère, l'indice -2 pour désigner l'avant dernier, ainsi de suite jusqu'à la valeur de `-len(chaîne)` pour désigner le premier caractère.

Exemple :

```
>>> s = "Informatique"
>>> s[-1]
'e'
>>> s[-2]
'u'
>>> s[-len(s)]
'I'
```

Memento

- l'accès à un caractère d'une chaîne se fait via son indice, noté entre crochets
- les indices vont de 0 inclus à `len(chaine)` exclu
- l'itération sur une chaîne peut se faire via ses indices
- l'accès à une tranche d'une chaîne se fait par un indiçage par un intervalle
- l'indiçage à partir de la fin de chaîne se fait avec des valeurs d'indices négatives

Codage des caractères et comparaison des chaînes de caractères

Coup d'œil

On découvre

- comment sont codés les caractères dans la mémoire de l'ordinateur
- le code ASCII et d'autres codes tels Unicode
- comment comparer des caractères
- l'ordre lexicographique
- comment comparer des chaînes de caractères

Codage des caractères

Les caractères sont des symboles qui peuvent être écrits.
Pour pouvoir être mémorisés et manipulés dans un ordinateur, ces caractères sont **codés** :

- on associe par exemple un entier à chaque caractère,
- ce sont ces entiers qui sont stockés dans la mémoire de l'ordinateur.

Un **code** est une convention ou une norme utilisée pour pouvoir travailler avec cette représentation des caractères.
Les codes les plus utilisés sont les codes ASCII et Unicode.

Code ASCII

La table de caractères ASCII (*American Standard Code for Information Interchange*) est l'une des plus anciennes utilisées en informatique.
Elle définit un jeu de 128 caractères numérotés de 0 à 127 :

code	char	code	char	code	char	code	char	code	char	code	char	code	char	code	char
0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

Dans ce jeu de caractères ASCII, les caractères de code compris entre 0 et 31, et le caractère de code 127 sont appelés caractères *de contrôle*.
On y trouve par exemple

- le caractère de code ASCII 9 : la tabulation, abrégé `ht` pour *horizontal tab*, et noté `\t` dans les chaînes de caractères
- le caractère de code ASCII 10 : le retour à la ligne, abrégé `nl` pour newline, et noté `\n` dans les chaînes de caractères

Les autres caractères, de code compris entre 32 et 126 sont les caractères dits *imprimables*.
On y trouve par exemple :

- le caractère de code ASCII 32 : l'espace, abrégé `sp`
- des caractères de ponctuations et d'autres symboles divers
- les 10 chiffres, de `0` de code 48, à `9` de code 57
- les 26 lettres de l'alphabet latin en version majuscules, de `A` de code 65, à `Z` de code 90
- les 26 lettres de l'alphabet latin en version minuscule, de `a` de code 97, à `z` de code 122

Autres codes, ISO 8859 et Unicode

Le code ASCII défini au début des années 1960 est limité. Ce code d'abord américain ne permet pas de représenter les lettres accentuées ou les lettres d'alphabets non latins.

Plusieurs extensions ont été proposées au fil des années pour prendre en compte une nécessaire internationalisation

- dans années 1985 les standards de la famille ISO 8859, comme le ISO 8859-1 – souvent appelée Latin-1 – qui intègre les lettres accentuées des langues de l'Europe occidentale
- à partir de 1990, le standard **Unicode** qui réunit en un même code l'ensemble des alphabets latin, cyrillique, grec, arabe, etc.

  Unicode est actuellement dans sa neuvième version. Ce code recense 128.172 caractères, et attribue à chacun d'eux un nom et un numéro.

  L'**UTF-8** est la méthode standard la plus connue pour le codage des caractères selon la norme Unicode. Elle est utilisée par la très grande majorité des pages web, et sur les systèmes GNU/Linux.

Il est essentiel de noter que ces codes ISO ou Unicode sont des **extensions du code ASCII** : ils reprennent les 127 codes ASCII et ajoutent d'autres caractères pour les codes supérieurs à 127.

Fonctions `ord()` et `chr()`

En Python, il est possible de consulter la table de codage des caractères, et en particulier la table ASCII, via les deux fonctions prédéfinies `ord()` et `chr()` :

- pour un caractère donné, `ord()` renvoie son code
- pour un code donné, `chr()` renvoie le caractère correspondant

Par exemple :

```
>>> chr(9)
'\t'
>>> ord('\n')
10
>>> ord('0')
48
>>> chr(65)
'A'
```

Ordre des caractères, comparaison de caractères

Un code comme l'ASCII définit un **ordre** sur les caractères : le caractère `0` de code 48 est inférieur au caractère `A` de code 65.

Cet ordre est utilisé par les opérateurs de comparaison Python <=, <, >=, et > appliqués aux caractères :

```
>>> '0' < 'A'
True
```

Les comparaisons de caractères qui ont le plus de sens sont celles comparant deux lettres minuscules ou deux lettres majuscules entre elles, l'ordre alphabétique (celui du dictionnaire) est alors respecté :

```
>>> 'a' <= 'e'
True
>>> 'A' <= 'E'
True
```

Par contre, toutes les lettres majuscules ont un code inférieur à toute lettre minuscule. Les comparaisons entre minuscules et majuscules sont donc de peu d'intérêt :

```
>>> 'a' <= 'A'
False
>>> 'E' <= 'a'
True
```

Comparaison de chaînes de caractères

Ordre lexicographique

L'ordre alphabétique permet d'ordonner les lettres et, par extension, les mots entre eux.
Pour ordonner des chaînes quelconques qui peuvent contenir des caractères autres que des lettres, il faut étendre l'ordre alphabétique. On parle d'**ordre lexicographique**.

L'ordre sur les caractères défini par le codage des caractères, par exemple ASCII, est ainsi étendu aux chaînes de caractères.

Comparaison de chaînes

En Python, les opérateurs de comparaison `<`, `<=`, `>`, et `>=` sont aussi utilisés pour comparer des chaînes de caractères.

La comparaison de deux chaînes procède ainsi :

- les chaînes sont comparées caractère par caractère, de gauche à droite
- le premier caractère de la première chaîne est comparé au premier caractère de la seconde chaîne, le second avec le second, etc.
- dès qu'un caractère est supérieur à un autre, la chaîne qui le contient est dite supérieure à l'autre et la comparaison s'arrête.

Lors de la comparaison de deux chaînes de différentes longueurs, la plus courte des deux peut être supérieure à l'autre. Par exemple :

```
>>> "Programme" < "Python"
True
```

parce que `'r' < 'y'`.

Lien avec l'ordre alphabétique

L'ordre sur les caractères et les chaînes de caractères ne respecte pas l'ordre alphabétique traditionnel :

- en raison de l'ordre défini entre les lettres majuscules et les lettres minuscules,
- en raison de l'ordre défini sur les lettres accentuées et autres signes.

Pour éviter les situations telles

```
>>> "programmation" < "programme"
True
>>> "programmation" < "Programme"
False
```

ou

```
>>> 'e' < 'i'
True
>>> 'é' < 'i'
False
```

ou encore

```
>>> "porte" < "porter"
True
>>> "porté" < "porter"
False
```

et

```
>>> "portemine" < "porte-savon"
False
>>> "portée" < "porte-savon"
False
```

On se limitera par exemple aux comparaisons de mots composés uniquement de lettres minuscules non accentuées, ou uniquement de lettres majuscules non accentuées.

Memento

- la table du code ASCII définit un code entier pour nos caractères les plus courants, mais pas les lettres accentuées
- la comparaison des caractères est basée sur ce code
- la comparaison de minuscules et majuscules est trompeuse
- l'ordre lexicographique étend à tous les caractères l'ordre alphabétique habituel
- les opérateurs Python de comparaison de chaînes <, <= et autres sont à manier avec précaution

Non mutabilité des chaînes de caractères

Coup d'œil

On découvre

- la notion de mutabilité
- comment modifier une chaîne à une position donnée
- comment modifier un caractère donné dans une chaîne

Mutabilité

Nous avons vu précédemment que les séquences sont des collections ordonnées d'éléments. Elles sont un cas particulier d'itérables car il existe des itérables non ordonnés, tels que les dictionnaires ou les ensembles que nous découvrirons plus tard.

Parmi les séquences, nous connaissons les intervalles (`range`) et les chaînes de caractères. Ces deux itérables ne sont pas mutables, c'est-à-dire que l'on ne peut pas en modifier les valeurs.

Il est logique de ne pas pouvoir modifier un élément d'un intervalle. Sinon, on modifierait la définition même de cet intervalle. En Python, contrairement à d'autres langages de programmation, on ne peut pas non plus modifier un caractère individuel d'une chaîne. En d'autres termes, on ne peut pas affecter une nouvelle valeur à un caractère donné d'une chaîne.

```
>>> s = "Informatique"
>>> s[0] = 'i'
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Le seul moyen de modifier une variable de type `str` est de créer une nouvelle variable qui reprend le contenu de l'ancienne, aux modifications près. Cette variable peut avoir le même nom que l'ancienne.

Modification d'une chaîne à une position donnée

Il s'agit de créer une nouvelle chaîne qui est la modification d'une chaîne à une position connue à l'avance.

Ajout de caractères

L'ajout d'un ou plusieurs caractères en début ou fin de chaîne se fait à l'aide d'une concaténation entre la chaîne d'origine s et les caractères à ajouter, placés devant ou derrière s.

```
>>> s = "ordinaire"
>>> prefixe = "extra"
>>> s = prefixe + s
>>> s
'extraordinaire'
>>> s = s + " !"
```

```
>>> s
'extraordinaire !'
```

L'ajout d'un ou plusieurs caractères à une position connue d'une chaîne se fait en découpant la chaîne de part et d'autre de la position d'insertion et en concaténant les sous-chaînes obtenues avec les caractères à ajouter, dans le bon ordre.

```
>>> s = "Il fait beau."
>>> s[0:4] + "ais" + s[4:len(s)]
'Il faisait beau.'
```

Suppression de caractères

La suppression d'un ou plusieurs caractères à des positions données d'une chaîne se fait en concaténant les sous-chaînes que l'on veut conserver.

```
>>> s = 'Il faisait beau.'
>>> s[0:6] + s[9:len(s)]
'Il fait beau.'
```

Remplacement de caractères

La remplacement de caractères revient à faire en même temps une suppression et un ajout.

```
>>> s = "Il fait beau."
>>> s[0:8] + "froid" + s[-1:len(s)]
'Il fait froid.'
```

Modification de caractères donnés d'une chaîne

Dans cette partie, la position des caractères à modifier n'est pas connue. Seuls les caractères à modifier le sont. Il est donc nécessaire de parcourir la chaîne afin de localiser les caractères à modifier.

Une nouvelle variable est initialisée vide avant l'entrée dans une boucle `for`. Cette boucle parcourt la valeur de la chaîne à modifier caractère par caractère et l'affecte à l'identificateur de boucle. Si le caractère pointé par l'identificateur est le caractère à modifier, la modification est répercutée dans la nouvelle variable. Sinon, ce caractère est ajouté à la nouvelle variable.

Ajout de caractères

L'exemple suivant illustre l'ajout systématique d'un ou plusieurs caractères avant et/ou après un caractère donné dans une chaîne. La fonction `entoure_espaces()` crée une nouvelle chaîne dans laquelle le caractère `*` est ajouté avant et après les espaces présents dans la chaîne passée en paramètre. Une nouvelle variable, *res*, est initialisée vide. Une boucle parcourt la valeur de *s* caractère par caractère et l'affecte à l'identificateur *carac*. Si le caractère pointé est une espace, `*` est ajouté avant et après *carac* et le tout est concaténé à *res*, sinon *carac* est concaténé seul à *res*.

```
def entoure_espaces(s):
    """
    s (str) -> s avec les espaces entourés de *
    Contrainte : aucune
    Exemples
    >>> entoure_espaces("valeur de la chaine") == 'valeur* *de* *la* *chaine'
    True
    >>> entoure_espaces("test") == "test"
    True
    >>> entoure_espaces("  ") == '* ** *'
    True
    """
    res = ""
    for carac in s:
        if carac == ' ':
            res = res + '*' + carac + '*'
        else:
```

```
            res = res + carac
    return res
```

Suppression de caractères

L'exemple suivant illustre la suppression systématique d'un caractère donné dans une chaîne. La fonction `supprime_espace()` crée une nouvelle chaîne dans laquelle tous les espaces sont supprimés de la chaîne *s* passée en paramètre. De la même manière, une boucle `for` parcourt la valeur de *s* caractère par caractère. Si *carac* n'est pas une espace, il est concaténé à *res*, sinon l'espace ne l'est pas. Ainsi, seuls les caractères autres que des espaces sont présents dans *res*.

```
def supprime_espaces(s):
    """
    s (str) -> s sans espace
    Contrainte : aucune
    Exemples
    >>> supprime_espaces("valeur de la chaine") == 'valeurdelachaine'
    True
    >>> supprime_espaces("test") == "test"
    True
    >>> supprime_espaces("  ") == ''
    True
    """
    res = ""
    for carac in s:
        if carac != ' ':
            res = res + carac
    return res
```

Modification de caractères

L'exemple suivant illustre la modification systématique d'un caractère donné dans une chaîne. Cette modification consiste à créer une nouvelle chaîne dans laquelle tous les espaces sont remplacés par le caractère `_`. La chaîne à modifier, *s* est passée en paramètre de la fonction `remplace_espaces()`. Dans la boucle parcourant *s*, si le caractère pointé est une espace, le caractère `_` est concaténé à *res*, sinon c'est *carac*. Ainsi, les espaces sont remplacés par `_`.

```
def remplace_espaces(s):
    """
    s (str) -> s avec les espaces remplacés par _
    Contrainte : aucune
    Exemples
    >>> remplace_espaces("valeur de la chaine") == 'valeur_de_la_chaine'
    True
    >>> remplace_espaces("test") == 'test'
    True
    >>> remplace_espaces("  ") == '__'
    True
    """
    res = ""
    for carac in s:
        if carac == ' ':
            res = res + '_'
        else:
            res = res + carac
    return res
```

Memento

- On ne peut pas modifier un élément d'une séquence non mutable
- `range()` et `str` sont non mutables
- pour modifier une chaîne à une position donnée, il est conseillé d'utiliser les sous-chaînes
- pour modifier des caractères donnés d'une chaîne, il est conseillé de parcourir la chaîne à l'aide d'une boucle `for` est de construire la chaîne modifiée à partir des caractères de la chaîne d'origine

Exercices niveau débutant

Indiçage

217. N'utilisez pas tout de suite l'interpréteur Python et anticipez les valeurs des expressions ci-dessous. Ensuite, vérifiez la validité de votre réponse en utilisant l'interpréteur.

```
s = "Bonjour"
s[1]
s[len(s)]
s[-1]
s[0:3]
s[3:len(s)-1]
```

Ordre lexicographique

218. Aidez-vous de la tables de caractères ASCII donnée en cours pour classer toutes les chaînes suivantes dans l'ordre croissant.

‘Bonjour’ ‘bonjour’ ’ bonjour’ ‘Bonjour.’ ‘Bonjour !’ ‘@univ’ ‘1’ ‘11’ ‘100’ ‘[10]’ ‘{10}’ ‘(10)’

219. Écrivez une fonction `maximum()` prenant en paramètre deux chaînes de caractères et renvoyant la plus grande de ces 2 chaînes.

220. Écrivez une fonction `minimum()` prenant en paramètre deux chaînes de caractères et renvoyant la plus petite de ces 2 chaînes.

221. Est-ce que le code de ces fonctions diffère de celui que vous avez écrit pour comparer des entiers dans les exercices sur les instructions conditionnelles ? Que pouvez-vous en déduire sur les types tolérés en paramètre de ces fonctions ?

Parcours d'une chaîne à l'aide de ses indices

222. Écrivez une fonction `miroir()` qui prend en paramètre une chaîne et qui renvoie la chaîne inversée, caractère par caractère. Cette fonction utilisera les indices pour parcourir la chaîne à l'envers.

```
>>> miroir('abc') == 'cba'
True
>>> miroir('') == ''
True
>>> miroir('a') == 'a'
True
>>> miroir('bonjour') == 'ruojnob'
True
>>> miroir('Vive Zorglub !') == '! bulgroZ eviV'
True
```

223. Écrivez une fonction `un_caractere_sur_deux()` qui prend en paramètre une chaîne et qui renvoie une chaîne constituée d'un caractère sur deux de la chaîne passée en paramètre, en gardant le premier.

```
>>> un_caractere_sur_deux('abc') == 'ac'
True
>>> un_caractere_sur_deux('') == ''
```

```
True
>>> un_caractere_sur_deux('a') == 'a'
True
>>> un_caractere_sur_deux('bonjour') == 'bnor'
True
>>> un_caractere_sur_deux('Vive Zorglub !') == 'Vv ogu '
True
```

224. Écrivez une fonction `dernier_indice_caractere()` qui renvoie l'indice de la dernière occurrence du caractère passé en paramètre, dans la chaîne passée en paramètre.
On considère que ce caractère est obligatoirement présent dans la chaîne.

```
>>> dernier_indice_caractere("Bonjour", "B") == 0
True
>>> dernier_indice_caractere("Bonjour", "o") == 4
True
>>> dernier_indice_caractere("Bonjour", "r") == 6
True
```

225. Écrivez une fonction `premier_indice_caractere()` qui renvoie l'indice de la première occurrence du caractère passé en paramètre, dans la chaîne passée en paramètre.
On considère que le caractère recherché est présent dans la chaîne.

```
>>> premier_indice_caractere("Bonjour", "B") == 0
True
>>> premier_indice_caractere("Bonjour", "o") == 1
True
>>> premier_indice_caractere("Bonjour", "r") == 6
True
```

Exercices niveau intermédiaire

Parcours de chaînes

Un palindrome est un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, ICI, ELLE et RADAR sont des palindromes. Dit autrement, les palindromes sont des mots ou des chaînes qui sont égales à leur version miroir.

226. Écrivez une première version du prédicat `est_palindrome()` prenant en paramètre une chaîne et renvoyant `True` si cette chaîne est un palindrome et `False` sinon. Cette version utilisera la fonction `miroir()` précédemment écrite.
227. Écrivez une deuxième version de ce prédicat qui compare la 1re lettre de la chaîne à la dernière, puis la 2e à l'avant dernière et ainsi de suite jusqu'à atteindre le milieu de la chaîne. Si ces paires de lettres sont toutes identiques, alors la chaîne est un palindrome.

Application aux séquences nucléiques

Les questions qui suivent sont une application concrète de l'utilisation de chaînes de caractères à une autre science, la biologie. Vous n'avez pas besoin de connaissances préalables en biologie pour y répondre.

Une séquences d'ADN peut être représentée par une chaîne ne contenant que les quatre lettres A, C, G et T, comme par exemple la chaîne :

ACGGTAGCTAGTTTCGACTGGAGGGGTA

Ces lettres représentent les nucléotides, c'est-à-dire les briques de base de la molécule d'ADN : A = adénine, T = thymine, G = guanine et C = cytosine.

Fonctions utiles pour commencer

228. Réalisez une fonction nommée `est_ADN()` qui vérifie si la chaîne passée en paramètre ne contient aucun autre caractère que les quatre bases A, C, G et T. Cette fonction retourne la valeur `True` si tel est le cas, et la valeur `False` dans le cas contraire. De plus, elle renvoie `True` si la chaîne est vide.

```
>>> est_ADN('ATGCGATC')
True
>>> est_ADN('ACKT')
False
>>> est_ADN('ACTK')
False
>>> est_ADN("")
True
```

Il est possible de générer aléatoirement une séquence ADN. La version naïve suppose que les 4 bases ont la même probabilité d'apparaître à une position donnée.

229. Réalisez une fonction nommée `genere_ADN()` qui renvoie une séquence ADN générée aléatoirement et dont la taille est passée en paramètre.

Un brin et son complémentaire

Dans les cellules vivantes, l'ADN est sous la forme double brin, c'est-à-dire que 2 séquences ADN se font face. Une séquence est lue de gauche à droite et l'autre de droite à gauche. De plus, les bases complémentaires l'une

de l'autre se font face. Par exemple, les 2 séquences suivantes sont complémentaires-inversées l'une de l'autre et peuvent ainsi former un double brin :

GTACA
TGTAC

Les bases A et T sont complémentaires entre elles, ainsi que les bases G et C.

230. Réalisez une fonction nommée `base_complementaire()` qui renvoie la base complémentaire de l'unique base passée en paramètre. Une contrainte d'utilisation de cette fonction est que son paramètre est bien une et une seule des quatre bases (`A`, `T`, `G` ou `C`).

```
>>> base_complementaire('G')
'C'
>>> base_complementaire('T')
'A'
```

231. Réalisez une fonction nommée `sequence_complementaire_inversee()` qui construit la séquence ADN complémentaire et inversée de celle passée en paramètre. Pour cela, cette fonction fera appel à la fonction `base_complementaire()`. Une contrainte d'utilisation de cette fonction est que son paramètre est bien une séquence ADN.

```
>>> sequence_complementaire_inversee('ACTG')
'CAGT'
```

Recherche de motifs

Certaines des enzymes qui interagissent avec l'ADN reconnaissent des motifs ADN, c'est-à-dire une suite de nucléotides.

232. Réalisez une fonction nommée `nombre_occurrences_motif()` qui compte le nombre fois où le motif passé en paramètre est présent dans la séquence d'ADN passée également en paramètre.

```
>>> nombre_occurrences_motif('ACG', 'GCTACGGAGCTTCGGAGCACGTAG')
2
>>> nombre_occurrences_motif('TTC', 'AGTCGACTT')
0
```

Modification de chaînes

233. Définissez une fonction `ajout_tirets()` qui renvoie la chaîne prise en paramètre avec des tirets `-` insérés derrière chacun de ses caractères.

```
>>> ajout_tirets("Bonjour") == "B-o-n-j-o-u-r-"
True
>>> ajout_tirets("a") == "a-"
True
```

234. Que se passe-t'il si vous passez en paramètre une chaîne vide ? Pourquoi ?

235. Définissez une deuxième version de cette fonction qui n'ajoute pas de tiret après le dernier caractère.

```
>>> ajout_tirets_v2("Bonjour") == "B-o-n-j-o-u-r"
True
>>> ajout_tirets_v2("a") == "a"
True
>>> ajout_tirets_v2("") == ""
True
```

236. Définissez une fonction `supprime_caractere()` prenant en paramètre une chaîne s de taille quelconque et un seul caractère c et renvoyant la chaîne s privée du caractère c.

```
>>> supprime_caractere("valeur de la chaine", " ") == 'valeurdelachaine'
True
>>> supprime_caractere("test", "t") == "es"
True
```

```
>>> supprime_caractere("test", "a") == "test"
True
```

237. Une phrase palindrome est une phrase qui forme un palidrome lorsque l'on enlève tous les espaces qui la compose. La chaîne obtenue n'a pas de sens, mais doit être palindromique. En utilisant certaines des fonctions codées précédemment, écrivez une fonction `est_phrase_palindrome()` qui renvoie `True` si la phrase passée en paramètre est une phrase palindrome. Par exemple, "eh ca va la vache" devient "ehcavalavache" sans les espaces et cette chaîne est un palindrome.

```
>>> est_phrase_palindrome("eh ca va la vache")
True
>>> est_phrase_palindrome("engage le jeu que je le gagne")
True
>>> est_phrase_palindrome("Engage le jeu que je le gagne.")
False
```

Exploitation du code ASCII

Les exercices suivant proposent d'écrire des fonctions déjà fournies par la bibliothèque Python.
L'objet des exercices est de travailler directement sur les chaînes de caractères, sans appels aux fonctions prédéfinies, en se basant sur les propriétés de la table ASCII.
On utilisera donc uniquement les deux fonctions reflétant cette table : `ord()` et `chr()`.

Chiffre et nombres

238. Proposez une fonction `chiffre` qui renvoie la valeur entière d'un chiffre donné sous la forme d'un caractère. Par exemple :

```
>>> chiffre('0')
0
>>> chiffre('4')
4
```

239. Proposez une fonction `entier()` qui renvoie la valeur entière d'une suite de chiffres donnée sous la forme d'une chaîne de caractères

```
>>> entier('3142')
3142
```

240. Proposez une évolution de la fonction `entier()` pour prendre en compte des nombres précédés d'une éventuel signe :

```
>>> entier('3142')
3142
>>> entier('-3142')
-3142
>>> entier('+3142')
3142
```

Minuscules et majuscules

241. Proposez prédicat `est_lettre_majuscule()` qui renvoie la valeur vrai si et seulement si le caractère fourni en paramètre est une lettre majuscule.

     Proposez de même un prédicat `est_lettre_minuscule()`

242. Proposez une fonction `lettre_majuscule()` qui renvoie la lettre majuscule d'une lettre minuscule donnée.

243. Proposez une évolution de la fonction `lettre_majuscule()` qui renvoie la lettre majuscule correspondante à y un caractère donné, ou ce caractère inchangé si ce n'est pas une lettre minuscule.

244. Proposez une évolution de la fonction `en_majuscule()` qui renvoie une chaîne passée en paramètre dans laquelle chacune des lettres minscules a été remplacée par la majuscule correspondante.

```
>>> en_majuscule('vive Python !')
'VIVE PYTHON !'
```

Exercices niveau confirmé

Expressions bien parenthésées

Dans ces exercices, le mot expression désigne une chaîne de caractères ne contenant que des parenthèses ouvrantes et fermantes comme par exemple "(()())", "(()()" et "(()))(".

Une expression est bien parenthésée

- si le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes,
- et si quelque soit la position dans l'expression, le nombre de parenthèses ouvrantes qui précèdent cette position est toujours supérieur ou égal au nombre de parenthèses fermantes qui précèdent.

Ainsi :

- "(()())" est une expression bien parenthésée
- "(()()" est mal parenthésée car il y a 3 parenthèses ouvrantes et seulement 2 parenthèses fermantes
- "(()))(" est mal parenthésée car le cinquième caractère est la troisième parenthèse fermante, alors qu'il n'y a que deux parenthèses ouvrantes qui précèdent

Bien parenthésée

Soit l'algorithme suivant :
Dans un parcours de gauche à droite d'une expression, on utilise un compteur qu'on incrémente à chaque fois qu'on rencontre une parenthèse ouvrante et qu'on décrémente à chaque fois qu'on rencontre une parenthèse fermante,

245. Indiquez à quelle condition sur les valeurs prises par ce compteur l'expression est bien parenthésée. Illustrez votre réponse avec les trois exemples ci-dessus.

246. Réalisez un prédicat `bien_parenthesee()` qui renvoie la valeur booléenne vrai si et seulement si la chaîne passée en paramètre est bien parenthésée.

Factorisation

Par la suite, on ne travaille qu'avec des expressions bien parenthésées.

On appelle factorisation d'une expression un découpage de cette expression en expressions bien parenthésées consécutives.
Chacune de ces expressions étant appelée facteur.

Voici quelques factorisations d'expressions bien parenthésées :

- "(()())" → "(()())" – un seul facteur
- "()(()())" → "()" + "(()())" – deux facteurs
- "(())()(()())" → "(())" + "()" + "(()())" – trois facteurs

247. Réalisez une fonction `nbre_facteurs()` calcule le nombre de facteurs que possède une expression bien parenthésée donnée.

248. Réalisez maintenant une fonction `decoupe_facteurs()` qui renvoie tous les facteurs d'une expression bien parenthésée en insérant entre eux le caractère '*' :

```
>>> decoupe_facteurs('(()())') == '(()())'
True
>>> decoupe_facteurs('(())()(()())') == '(())*()*(()())'
True
```

Modification de chaînes

Censure (épisode 1)

On souhaite pouvoir censurer une sous-chaîne dans un texte donné.

```
>>> censure1('La fonction `print()` peut être pratique.', 'print')
'La fonction `*****()` peut être pratique.'
>>> censure1('Bal tragique à Colombey : 1 mort', 'o')
'Bal tragique à C*l*mbey : 1 m*rt'
```

Définissez une fonction `censure1()`, paramétrée par deux chaînes, et renvoyant le texte de la première, en remplaçant chacune des occurences de la seconde par autant d'étoiles `*` que la *longueur* de la seconde.

8. Itération conditionnelle - while

Itération conditionnelle

Coup d'œil

On découvre

- comment répéter l'exécution de certaines parties d'un programme tant qu'une condition est respectée
- ce qu'est une itération conditionnelle

Itération conditionnelle

Pour répéter une séquence d'instructions, il est possible dans certains cas d'utiliser un itérable et une boucle `for` si l'on connait à l'avance le nombre d'itérations de la boucle.

On peut aussi vouloir répéter une séquence d'instructions tant qu'une certaine condition est – ou n'est pas – vérifiée.
Le nombre d'itérations n'est alors pas connu à l'avance.
Par exemple une recherche de seuil comme : « à partir de quelle valeur la somme des premiers entiers $1 + 2 + 3 + \cdots + n$ dépasse 1000 ? ».

Il nous faut pouvoir identifier :

- la condition : ce sera une valeur booléenne.
- la ou les instructions à répéter : ce sera un bloc d'instructions, repéré par une indentation supplémentaire.

Dans notre exemple :

- la condition sera que la somme ne dépasse pas 1000
- les instructions à répéter devront mettre à jour la valeur de n et la somme

Une telle structure est une boucle appelée *itération conditionnelle.*

Instruction `while`

En Python, on utilisera le mot clé `while` et on écrira par exemple :

```
n = 1
somme = 1
while somme <= 1000 :
    n = n + 1
    somme = somme + n
```

Que l'on peut lire : « tant que la somme est inférieure à 1000, ajouter 1 à `n` et ajouter `n` à `somme` ».

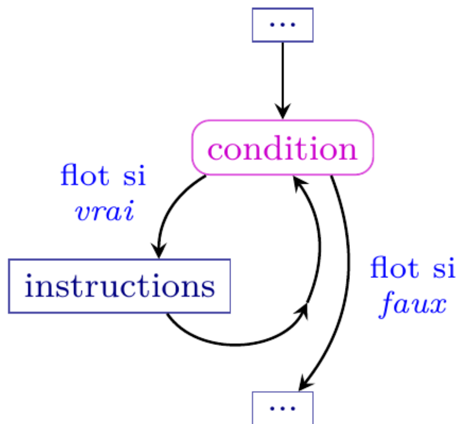
Flot d'exécution

Rappelons que le flot d'exécution est l'ordre dans lequel les instructions sont exécutées.

Considérons le code suivant :

```
...
while condition :
    instructions
...
```

Le flot d'exécution correspondant peut être illustré par le graphe :



- la condition est évaluée. On obtient une valeur booléenne.
- si cette valeur est *vrai*
    - les instructions associées au `while` sont exécutées
    - on retourne au point précédent pour évaluer à nouveau la condition
- si cette valeur est *faux*
    - on passe à l'exécution des instructions suivantes (...).

Exemple complémentaire

Des épidémiologistes estiment que le nombre de cas positifs double tous les trois jours.
Des médecins connaissent le nombre de cas positifs actuels, et voudraient calculer le temps au bout duquel ce nombre de cas positifs atteindra une valeur donnée.

Par exemple pour une population contenant 100 cas positifs :

- au bout de 3 jours, il y a aura 200 cas positifs,
- au bout de 6 jours, il y aura 400 cas positifs, et
- au bout de 9 jours il y en aura 800.

On dépassera alors, par exemple :

- le seuil de 400 au bout de 6 jours,
- le seuil de 401 au bout de 9 jours, et
- le seuil de 799 au bout de 9 jours également.

Pour calculer cette durée, écrivons une fonction `duree()` paramétrée par deux entiers : `cas_initiaux` représentant le nombre de cas positifs au jour 0, `seuil` représentant le seuil atteint.

Dans cette fonction, nous utiliserons deux variables :

- `jours` pour compter le nombre de jours,
- `cas` pour compter le nombre de cas.

Ces deux variables seront initialisées respectivement à `0` et `cas_initiaux`.

Les instructions qui doivent être répétées sont l'incrémentation de `jours` par 3, et le doublement de `cas`.

La condition devant être respectée pour continuer les itérations est `cas < seuil` : tant que le seuil n'est pas atteint ou dépassé, on continue de compter les jours.

```
def duree(cas_initiaux, seuil):
    """Renvoie le temps requis pour qu'une population de cas_initiaux cas
    positifs atteigne le seuil en doublant tous les trois jours.
    Paramètres :
     - cas_initiaux : int, nombre de cas positifs au jour 0
     - seuil : int, nombre de cas positifs à atteindre ou dépasser
    Valeur de retour : int, temps exprimé en nombre de jours
    Exemples :
    >>> duree(100, 400)
    6
```

```
    >>> duree(100, 401)
    9
    >>> duree(100, 799)
    9
    """
    jours = 0
    cas = cas_initiaux
    while cas < seuil :
        jours = jours + 3
        cas = cas * 2
    return jours
```

Remarques et compléments

Évolution de la valeur de la condition

Observons le code suivant :

```
# (1)
while condition :
    # (2)
    instructions
    # (3)
# (4)
```

dans lequel les commentaires `(1)`, `(2)`, `(3)` et `(4)` servent à indiquer un emplacement dans le flot d'exécution.

1. Juste avant la boucle – au point `(1)` –, l'expression booléenne `condition` peut prendre les valeurs
   - `True`, auquel cas on est sûr qu'il y aura au moins une itération, ou
   - `False`, auquel cas on est sûr qu'il n'y aura aucune itération.
2. Au début d'une itération – au point `(2)` –, l'expression booléenne `condition` est nécessairement vraie.
3. À la fin d'une itération – au point `(3)`–, l'expression booléenne `condition` peut valoir
   - `True`, auquel cas la boucle continuera pour une autre itération au moins, ou
   - `False`, auquel cas, la boucle s'arrêtera.
4. Juste après la boucle – au point `(4)` –, l'expression booléenne `condition` est nécessaire fausse.

Boucle infinie

Le nombre d'itérations d'une boucle `while` n'est a priori pas connu.

Il peut y avoir aucune itération : les instructions ne seront pas exécutées.

Il peut aussi arriver que la condition de la boucle `while` soit toujours vérifiée.
On parle de boucle infinie.

L'exécution du code ne termine pas…

```
i = 1
while i > 0 :
    i = i + 1
```

Boucle imbriquée

Nous avons vu que les boucles `for` peuvent être imbriquées sans nuire à la lisibilité du code.

Dans le cadre de ce cours, on évitera d'imbriquer une boucle `while` dans une autre boucle.

Si nécessaire, on préfèrera appeler une fonction auxiliaire – contenant une boucle `while` – dans une autre boucle.

Memento

- l’instruction d’itération conditionnelle `while` permet d’exécuter un bloc d’instructions tant qu’une condition est vérifiée

Utilisations typiques de boucles ‘while‘

Coup d’œil

On découvre

- quelques problèmes courants dont la résolution passe par l’utilisation d’itérations conditionnelles

Appartenance

On cherche à savoir si un caractère `c` appartient à une chaîne `s`.

Pour cela, on parcourt la chaîne, et on s’arrête une fois le caractère trouvé.
Il est inutile de continuer le parcours de la chaîne une fois le caractère trouvé.

Le nombre d’itérations ne peut pas être connu à l’avance. Nous allons donc utiliser une boucle `while`.

Fonction `appartient()`

Le principe est le suivant :

- on utilise un indice `i` commençant à 0, qui parcourra les indices des caractères de la chaîne
- on ne parcourra pas les caractères après que le caractère `c` soit trouvé.

La condition d’arrêt de la boucle sera alors : « le parcours de la chaîne n’est pas fini **et** le caractère `c` n’a pas été trouvé ».

Une fois la boucle terminée, on se pose la question de l’arrêt : la fin de la boucle implique

- soit que la chaîne a été parcourue complètement, et dans ce cas `i == len(s)` est vraie,
  Le caractère `c` n’appartient pas à la chaîne `s`.
- soit que `c` a été trouvé, et dans ce cas `i < len(s)` **et** `s[i] == c` sont vraies.
  Le caractère `c` appartient à la chaîne `s`.

La valeur de l’expression `i < len()` permet de déterminer ce pourquoi la boucle s’est terminée, et de conclure sur l’appartenance ou non du caractère `c` à la chaîne `s`.

On obtient la fonction suivante :

```
def appartient(c, s):
    """Renvoie True ssi la chaîne s contient le caractère c.
    Paramètres :
    - c (str) : caractère recherché
    - s (str) : chaîne dans laquelle on recherche c
    Valeur de retour (bool) : c appartient à s
    Contraintes : len(c) == 1
    Exemples :
    >>> appartient('e', 'panier')
    True
    >>> appartient('o', 'desert')
    False
    """
    i = 0
```

```
    while i < len(s) and s[i] != c:
        i = i + 1
    return i < len(s)
```

Exemples d'exécution

Exécutons cette fonction sur deux exemples simples.

```
>>> appartient('a', 'bac')
True
```

On initialise la variable `i` à la valeur 0. Cela signifie qu'on s'apprête à regarder la valeur de `s[0]`, le premièr caractère de la chaîne `s`.

La boucle `while` commence :

- on évalue l'expression booléenne : `0 < len(s)` est vraie, puis `s[0] != 'a'` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `1`.
- on évalue l'expression booléenne : `1 < len(s)` est vraie, puis `s[1] != 'a'` est fausse, la conjonction est donc fausse. La boucle s'arrête.

On évalue alors `1 < len(s)`. Comme cette expression est vraie, on renvoie `True`.

```
>>> appartient('a', 'bcd')
False
```

On initialise la variable `i` à la valeur 0. Cela signifie qu'on s'apprête à regarder la valeur de `s[0]`, le premier caractère de la chaîne `s`.

La boucle `while` commence :

- on évalue l'expression booléenne : `0 < len(s)` est vraie, puis `s[0] != 'a'` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `1`.
- on évalue l'expression booléenne : `1 < len(s)` est vraie, puis `s[1] != 'a'` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `2`.
- on évalue l'expression booléenne : `2 < len(s)` est vraie, puis `s[2] != 'a'` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `3`.
- on évalue l'expression booléenne : `3 < len(s)` est fausse, et on n'a donc pas besoin d'évaluer `s[3] != c` (cette évaluation déclencherait d'ailleurs une erreur d'indexation car `s[3]` n'existe pas).

On évalue alors `3 < len(s)`. Comme cette expression est fausse, on renvoie `False`.

Exemple d'utilisation de la fonction `appartient()`

Le prédicat `est_lettre_francaise()` qui permet de savoir si un caractère est une des quarante-deux lettres de l'alphabet français pourrait être défini en utilisant le prédicat `appartient()`.

(*au sujet des 42 lettres de l'alphabet français, voir par exemple la page `fr.wikipedia.org/wiki/Alphabet_français`*)

```
def est_lettre_francaise(c):
    """Renvoie True si c est une lettre française (peut importe la casse) et
    False sinon.
    Paramètres :
    - c (str) : le caractère à tester
    Valeur de retour (bool) : c est une lettre française
    Contraintes : len(c) == 1
    Exemples :
    >>> est_lettre_francaise('É')
    True
    >>> est_lettre_francaise('Ω')
    False
    """
```

```
    return (c >= 'a' and c <= 'z' or c >= 'A' and c <= 'Z'
           or appartient(c, 'ÀàÂâÆæÇçÉéÈèÊêËëÎîÏïÔôŒœÙùÛûÜüŸÿ'))
```

Indice

On cherche maintenant l'indice d'un caractère `c` dans une chaîne `s`.

Ce problème est en fait mal défini :

- que se passe-t-il dans une chaîne ne contenant pas `c` ?
- que se passe-t-il dans une chaîne contenant plusieurs fois `c` ?

Pour répondre à ces ambiguïtés, on précise le problème : on cherche

- le plus petit indice d'un caractère `c` dans une chaîne `s` (donc l'indice de la première occurrence de `c`),
- ou la taille de `s` si cette chaîne ne contient pas `c`.

Le code d'une fonction solution de ce problème est très similaire à celui du prédicat `appartient()` :

```
def indice(c, s):
    """Renvoie l'indice de première occurrence du caractère c dans la chaîne s si
    cette dernière contient c, len(s) sinon.
    Paramètres :
    - c (str) : caractère recherché
    - s (str) : chaîne dans laquelle on recherche c
    Valeur de retour (int) : indice de 1ère occurrence de c ou len(s)
    Contraintes : len(c) == 1
    Exemples :
    >>> indice('b', 'abba') == 1
    True
    >>> indice('o', 'desert') == len('desert')
    True
    """
    i = 0
    while i < len(s) and s[i] != c:
        i = i + 1
    return i
```

Pour tous

On cherche à vérifier si tous les éléments d'un itérable vérifient une propriété.

On cherche par exemple à savoir si tous les caractères d'une chaîne sont des lettres françaises.

Comme cela ne sert à rien de continuer de parcourir la chaîne si on a découvert un caractère qui n'est pas une lettre française, on utilisera une boucle `while`.

```
def que_des_lettres_francaises(s):
    """Renvoie True si tous les caractères de s sont des lettres françaises, et
    False sinon.
    Paramètres :
    - s (str)
    Valeur de retour (bool)
    Exemples :
    >>> que_des_lettres_francaises('œdème')
    True
    >>> que_des_lettres_francaises('top50')
    False
    """
    i = 0
    while i < len(s) and est_lettre_francaise(s[i]):
        i = i + 1
    return i == len(s)
```

Exemples d'exécution

Exécutons cette fonction sur deux exemples simples.

```
>>> que_des_lettres_francaises('H2O')
False
```

On initialise la variable `i` à la valeur 0. Cela signifie qu'on s'apprête à regarder la valeur de `s[0]`, le premier caractère de la chaîne `s`.

La boucle `while` commence :

- on évalue l'expression booléenne : `0 < len(s)` est vraie, puis `est_lettre_francaise(s[0])` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `1`.
- on évalue l'expression booléenne : `1 < len(s)` est vraie, puis `est_lettre_francaise(s[1])` est fausse, la conjonction est donc fausse, et la boucle s'arrête.

On évalue alors `1 == len(s)`. Comme cette expression est fausse, on renvoie `False`.

```
>>> que_des_lettres_francaises('été')
True
```

On initialise la variable `i` à la valeur 0. Cela signifie qu'on s'apprête à regarder la valeur de `s[0]`, le premier caractère de la chaîne `s`.

La boucle `while` commence :

- on évalue l'expression booléenne : `0 < len(s)` est vraie, puis `est_lettre_francaise(s[0])` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `1`.
- on évalue l'expression booléenne : `1 < len(s)` est vraie, puis `est_lettre_francaise(s[1])` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `2`.
- on évalue l'expression booléenne : `2 < len(s)` est vraie, puis `est_lettre_francaise(s[2])` est vraie, la conjonction est donc vraie.
  On exécute alors l'instruction `i = i + 1`, et `i` prend alors la valeur `3`.
- on évalue l'expression booléenne : `3 < len(s)` est fausse, et on n'a donc pas besoin d'évaluer `est_lettre_francaise(s[3])` (cette expression déclencherait d'ailleurs une erreur d'indexation car `s[3]` n'existe pas).

On évalue alors `3 == len(s)`. Comme cette expression est vraie, on renvoie `True`.

Autres utilisations typiques de boucles ‘while‘

Coup d’œil

On découvre

- d’autres problèmes courants dont la résolution passe par l’utilisation d’itérations conditionnelles

Suite de Syracuse

D’après `fr.wikipedia.org/wiki/Conjecture_de_Syracuse` :

> En mathématiques, on appelle suite de Syracuse une suite d’entiers naturels définie de la manière suivante : on part d’un nombre entier plus grand que zéro ;
>
> - s’il est pair, on le divise par 2 ;
> - s’il est impair, on le multiplie par 3 et on ajoute 1.
>
> En répétant l’opération, on obtient une suite d’entiers positifs dont chacun ne dépend que de son prédécesseur.
>
> Par exemple, à partir de 14, on construit la suite des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2…
> C’est ce qu’on appelle la suite de Syracuse du nombre 14.
>
> Après que le nombre 1 a été atteint, la suite des valeurs (1, 4, 2, 1, 4, 2…) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial.
>
> On n’a jamais trouvé d’exemple de suite obtenue suivant les règles données qui n’aboutisse pas à 1, puis au cycle trivial.

On cherche à connaître la *durée du vol* de Syracuse d’un nombre strictement positif n, c’est-à-dire le nombre d’étapes nécessaires pour atteindre 1.
Par exemple, pour 14, il faudra 17 étapes.

Le nombre d’étapes n’est pas connu a priori (c’est d’ailleurs la valeur que l’on cherche à calculer !). On utilisera donc une boucle `while`.

```
def vol_syracuse(n):
    """Nombre d'étapes nécessaires pour atteindre 1 en partant de n
    dans la suite de Syracuse.
    Paramètres :
    - n (int)
    Valeur de retour (int) : nombre d'étapes
    Contraintes : n > 0
    Exemples
    >>> vol_syracuse(14) == 17
    True
    >>> vol_syracuse(1) == 0
    True
    """
```

```python
    etape = 0
    terme = n
    while terme != 1:
        if terme % 2 == 0:
            terme = terme // 2
        else:
            terme = terme * 3 + 1
        etape = etape + 1
    return etape
```

Comme des lapins !

Un couple de lapereaux est déposé sur une île.

1. Au bout du premier mois, ces lapins deviennent adultes et peuvent se reproduire. Il y a donc 1 couple de lapins adultes sur l'île, et 0 couple de jeunes.
2. Au bout du deuxième mois, ils donnent naissance à un couple de jeunes et continuent de se reproduire. Il y a donc 1 couples d'adultes sur l'île, et 1 couple de jeunes.
3. Au bout du troisième mois, le couple d'adultes encore naissance à des jeunes et continuent de se reproduire. Le couple de jeunes devient un couple d'adultes et commencent à se reproduire. Il y a donc 2 couples d'adultes et 1 couple de jeunes.
4. Au bout du quatrième mois, les deux couples d'adultes donnent chacun naissance à un couple de jeunes, et le couple de jeunes devient 1 couple d'adultes. Il y a donc 3 couples d'adultes et 2 couple de jeunes.

Et ça continue !

Chaque mois, tous les couples de jeunes deviennent un couple d'adultes, et tous les couples d'adultes donnent chacun naissance à un couple de jeunes. On se demande au bout de combien de mois ce nombre de couples de lapins dépassera un certain seuil.

Nous allons résoudre ce problème en utilisant une boucle dans laquelle chaque itération représentera un mois.

```python
def population_lapins(seuil):
    """Renvoie le nombre de mois au bout duquel le nombre de couples de lapins
    dépassera le `seuil`.
    Paramètres :
    - seuil (int) : seuil à dépasser (exprimé en nombre de couples de lapins)
    Valeur de retour (int) : nombre de mois
    Contraintes : seuil ≥ 0
    >>> population_lapins(4)
    4
    """
    # Au tout début, il y a 1 couple de jeunes et 0 couple d'adultes
    mois = 0
    couples_jeunes = 1
    couples_adultes = 0
    while couples_jeunes + couples_adultes < seuil:
        # les jeunes deviennent adultes
        couples_ados = couples_jeunes
        couples_jeunes = 0
        # les adultes donnent naissance à des jeunes
        couples_jeunes = couples_adultes
        # les ados rejoignent les adultes
        couples_adultes = couples_adultes + couples_ados
        # un mois s'est écoulé
        mois = mois + 1
    return mois
```

L'évolution de cette population de lapins suit la célèbre *suite de Fibonacci*.

PGCD

Pour calculer le plus grand diviseur commun entre deux nombres naturels, on peut utiliser l'algorithme d'Euclide, basé sur les deux égalités suivantes :

$$\forall (a,b) \in \mathbb{N} \quad pgcd(a,b) = \begin{cases} a & \text{si } b = 0 \\ pgcd(b, a - kb) & \forall k \text{ sinon} \end{cases}$$

TODO l'introduction de k est-elle utile ? Donner les égalités pour la seule valeur de k considérée ?

On prendra ici k le quotient de a dans la division euclidienne par b (et donc $a - kb$ vaudra le reste de a dans la division euclidienne par b).

Comme on ne sait pas combien d'itérations seront nécessaires, on utilise une boucle `while`.

```
def pgcd(x, y):
    """Renvoie le plus grand diviseur commun de x et y
    Paramètres :
    - x (int)
    - y (int)
    Valeur de retour (int) : le pgcd de x et y
    Contraintes : x ≥ 0 et y ≥ 0 et (x,y)≠(0,0)
    Exemples :
    >>> pgcd(7, 5) == 1
    True
    >>> pgcd(42, 2020) == 2
    True
    """
    # a, b = x, y
    a = x
    b = y
    while b > 0:
        # a, b = b, a%b
        reste = a % b
        a = b
        b = reste
    return a
```

Observons le comportement de cette fonction sur un exemple simple.

```
>>> pgcd(51, 42)
3
```

Nous allons représenter dans un tableau les différentes valeurs prises par les variables `a` et `b` juste avant la boucle, puis en fin de chaque itération.

a	b
51	42
42	9
9	6
6	3
3	0

Si on applique l'algorithme d'Euclide à la main, on obtient :

$$\begin{array}{rcl} 51 & = & 42 \times 1 + 9 \\ 42 & = & 9 \times 4 + 6 \\ 9 & = & 6 \times 1 + 3 \\ 6 & = & 3 \times 2 + 0 \end{array}$$

On remarque alors que chaque ligne (sauf la dernière) du tableau des valeurs correspond à une étape dans l'exécution de l'algorithme d'Euclide. `a` correspond alors au dividende et `b` au diviseur.

Exercices niveau débutant

Quotient et reste dans la division euclidienne

Observez la procédure suivante :

```
def divmod_verbeux(a, b):
    """Affiche les calculs qu'un élève de CE1 ferait pour
    obtenir le quotient et le reste de la division euclidienne
    de a par b.
    Contraintes : a >= 0 et b > 0
    """
    reste = a
    nb_soustractions = 0
    print("Au début, on a fait", nb_soustractions, "soustractions,",
          "et le reste est", reste)
    # début de la boucle
    while reste >= b:
        nb_soustractions = nb_soustractions + 1
        reste = reste - b
        print("Après", nb_soustractions, "soustractions, le reste vaut", reste)
    # fin de la boucle
    print("Au final, on a fait", nb_soustractions, "soustractions,",
          "et le reste est", reste)
```

249. Créez une procédure `main()` qui appelle `divmod_verbeux()` sur différentes valeurs de `a` et `b` respectant les contraintes.

     Testez votre programme affichant dans la console les calculs effectués pour la division euclidienne de 42 par 5.

     Que pouvez-vous dire de la condition `reste >= b` avant l'exécution de la boucle ? Après l'exécution de la boucle ?

250. Définissez une fonction `reste()` renvoyant le reste dans la division euclidienne du premier paramètre par le second. Cette fonction ne devra pas utiliser les opérateurs `//` et `%`.

251. Définissez une fonction `quotient()` renvoyant le quotient dans la division euclidienne du premier paramètre par le second. Cette fonction ne devra pas utiliser les opérateurs `//` et `%`.

Le blob

Le blob (*physarum polycephalum*) est un organisme mono-cellulaire qui double de taille tous les jours. On veut connaitre le nombre de jours nécessaires pour atteindre une taille de 1m.

252. Définissez une fonction `croissance_blob()` paramétrée par un flottant représentant la taille initiale en mètres, et renvoyant le nombre de jours après lesquels le blob atteindra ou dépassera 1 mètre.

Période radioactive

La *période radioactive* ou *demi-vie* d'un isotope est le temps nécessaire à la désintégration de la moitié des noyaux d'un fragment de cet isotope.

On cherche à calculer le nombre de périodes radioactives nécessaire à la disparition d'une partie des noyaux d'un isotope. Par exemple, pour faire disparaitre les trois quarts des noyaux d'un isotope donné, il faut attendre deux périodes radioactives. En effet, au bout d'une période, il en reste la moitié, et au bout de deux périodes il en reste un quart. Les trois quarts ont donc disparu…

253. Définissez une fonction `nb_periodes_rad()` paramétrée par un flottant représentant la part de noyaux d'un isotope que l'on veut voir disparaître et qui renvoie le nombre de périodes radioactive nécessaires à cette disparition.

Racine carrée entière

La *racine carrée entière* d'un nombre entier x naturel est le plus grand entier vérifiant $n^2 \leq x$. Si on nomme r cette racine carrée entière, alors r vérifie $r^2 \leq x < (r+1)^2$.

Pour calculer cette racine carrée entière, on teste consécutivement les entiers n en commençant à 0, en s'arrêtant quand n^2 dépasse x.

254. Définissez une fonction `racine_carree_entiere()` qui renvoie la racine carrée entière du nombre passé en paramètre.

```
>>> racine_carree_entiere(9)
3
>>> racine_carree_entiere(8)
2
>>> racine_carree_entiere(10)
3
```

Retour sur des exercices précédents

Les semaines précédentes, vous avez proposé certaines solutions d'exercices faisant des itérations inutiles.

Vous pourrez ici corriger ces fonctions en utilisant une boucle `while` à la place d'une boucle `for`.

255. Proposez un prédicat `sont_minuscules()` qui vérifie que tous les caractères d'une chaîne de caractères sont des lettres minuscules ou des espaces. On utilisera la fonction Python prédéfinie `islower()` pour vérifier qu'un caractère est une lettre minuscule :

256. Écrivez une troisième version du prédicat `est_palindrome()` qui vérifie si la chaîne passée en paramètre est un palindrome. Cette fonction renvoie la valeur `True` si tel est le cas, et la valeur `False` dans le cas contraire.

257. Réalisez une fonction nommée `est_ADN()` qui vérifie si la chaîne passée en paramètre ne contient aucun autre caractère que les quatre bases A, C, G et T. Cette fonction renvoie la valeur `True` si tel est le cas, et la valeur `False` dans le cas contraire. De plus, elle renvoie `True` si la chaîne est vide.

```
>>> est_ADN('ATGCGATC')
True
>>> est_ADN('ACKT')
False
>>> est_ADN('ACTK')
False
>>> est_ADN("")
True
```

Vol de Syracuse

Dans le cours on a donné la solution pour déterminer à quel rang la suite de Syracuse atteint 1.

258. Définissez une fonction `altitude_max_syracuse()` qui donne la valeur maximale durant la durée du vol (cette valeur est appelée altitude maximale).

Utilisez par exemple les exécutions suivantes pour tester votre fonction :

```
>>> altitude_max_syracuse(1024)
1024
```

```
>>> altitude_max_syracuse(871)
190996
```

Exercices niveau intermédiaire

Test de primalité naïf

Un entier p est dit premier si il est supérieur ou égal à 2 et si aucun des entiers compris entre 2 et $p-1$ inclus ne divise p.

259. Définissez un prédicat nommé `est_premier_naif()` renvoyant `True` si l'entier passé en paramètre est un nombre premier, et `False` sinon.

```
def est_premier_naif(p):
    """Renvoie True ssi p est premier.
    Paramètres :
    - p (int) : entier dont on veut tester la primalité
    Valeur de retour (bool) : primalité de p
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> est_premier_naif(1)
    False
    >>> est_premier_naif(13)
    True
    >>> est_premier_naif(2020)
    False
    """
```

Les intervalles peuvent aussi être indexés, comme les chaînes.

```
>>> range(2, 6)[1]
3
```

260. Définissez un prédicat `au_moins_un_premier` paramétré par un intervalle, et renvoyant `True` si et seulement si l'intervalle passé en paramètre contient au moins un nombre premier. Par exemple l'intervalle `range(12)` contient 2, qui est premier, alors qu'aucun des trois éléments de l'intervalle `range(8, 11)` (qui sont 8, 9 et 10) n'est premier.

```
def au_moins_un_premier(intervalle):
    """Renvoie True si intervalle contient au moins un nombre premier.
    Paramètres :
    - intervalle (range)
    Valeur de retour (bool)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> au_moins_un_premier(range(12))
    True
    >>> au_moins_un_premier(range(8,11))
    False
    """
```

Écriture décimale d'un nombre

Dans tout cet exercice, il sera interdit d'utiliser `str()`.

261. Définir une fonction `somme_chiffres()` renvoyant la somme des chiffres dans l'écriture décimale du nombre passé en paramètre.

    Voici comment pourrait marcher l'algorithme sur 2021 :

    1. On initialise une variable `somme` à 0 et une autre variable `reste` à 2021.
    2. On incrémente la somme du reste dans la division euclidienne de 2021 par 10 (`somme` vaut donc 1) et on change la valeur de `reste` en le quotient dans la division euclidienne de 2021 par 10 (`reste` vaut donc 202).
    3. On incrémente la somme du reste dans la division euclidienne de 202 par 10 (`somme` vaut donc 3) et on change la valeur de `reste` en le quotient dans la division euclidienne de 202 par 10 (`reste` vaut donc 20).
    4. On incrémente la somme du reste dans la division euclidienne de 20 par 10 (`somme` vaut donc 3) et on change la valeur de `reste` en le quotient dans la division euclidienne de 20 par 10 (`reste` vaut donc 2).
    5. On incrémente la somme du reste dans la division euclidienne de 2 par 10 (`somme` vaut donc 5) et on change la valeur de `reste` en le quotient dans la division euclidienne de 2 par 10 (`reste` vaut donc 0).
    6. Comme `reste` vaut 0, on s'arrête et on renvoie `somme`

```python
def somme_chiffres(n):
    """Renvoie la somme des chiffres de n
    Paramètres :
    - n (int)
    Valeur de retour (int) : somme des chiffres de n
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> somme_chiffres(2021) == 5
    True
    """
```

262. Définir une fonction `miroir_chiffres()` renvoyant le nombre dont l'écriture décimale est la symétrique de celle du nombre passé en paramètre. Par exemple `miroir_chiffres(2021)` a pour valeur `1202`.

```python
def miroir_chiffres(n):
    """Renvoie le nombre dont l'écriture est la symétrique de celle de n
    Paramètres :
    - n (int)
    Valeur de retour (int)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> miroir_chiffres(2021) == 1202
    True
    >>> miroir_chiffres(2020) == 202
    """
```

Sous-chaînes

On dit que la chaîne `p` est *préfixe* de la chaîne `s` s'il existe une chaîne `w` telle que `s == p + w`, autrement si la chaîne `s` commence par la chaîne `p`. Remarquez que `''` la chaîne vide est préfixe de toute chaine (`w = s` convient), et toute chaîne `s` est préfixe d'elle-même (`w = ''` convient).

263. Définissez un prédicat `est_prefixe()` renvoyant `True` si son premier paramètre est préfixe du second et `False` sinon.

```python
def est_prefixe(p, s):
    """Renvoie True ssi p est préfixe de s.
    Paramètres :
    - p (str)
    - s (str)
    Valeur de retour (bool)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> est_prefixe("bon", "bonjour")
```

```
    True
    >>> est_prefixe("bonjour", "bon")
    False
    >>> est_prefixe("cuba", "cacao")
    False
    >>> est_prefixe("bon", "bon")
    True
    """
```

264. Définissez un prédicat `est_souschaine()` renvoyant `True` si son premier paramètre est sous-chaîne du second, et `False` sinon.

```
def est_souschaine(sc, s):
    """Renvoie True ssi sc est préfixe de s.
    Paramètres :
    - sc (str)
    - s (str)
    Valeur de retour (bool)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> est_souschaine("lit", "ponctualité")
    True
    >>> est_souschaine("pilote", "avion")
    False
    >>> est_souschaine("fin", "Début de la fin")
    True
    """
```

Mots de Dyck

On dit qu'une chaîne est *bien parenthésée* si à chaque parenthèse ouvrante est associée une parenthèse fermante plus loin dans la chaîne.

Par exemple `(()())` est bien parenthésée, par contre `())((()`, `()(` et `())` ne le sont pas.

265. À la main, définissez un compteur initialisé à 0, et parcourez la chaîne. À chaque occurrence de parenthèse ouvrante, incrémentez ce compteur, et décrémentez-le à chaque occurrence de parenthèse fermante.

     Appliquez cet algorithme sur les 4 exemples précédents. À quelles conditions sur ce compteur au cours du parcours de la chaîne et à la fin du parcours une chaîne est-elle mal parenthésée ? *(Indication : si vous n'avez pas trouvé ces conditions, vous pouvez consulter l'exercice sur les expressions bien parenthésées dans la partie avancée des exercices sur les chaînes.)*

266. Définissez un prédicat `est_bien_parenthesee()` qui renvoie `True` si la chaîne passée en paramètre est bien parenthésée et `False` sinon.

```
def est_bien_parenthesee(s):
    """Renvoie True ssi s est bien parenthésée.
    Paramètres :
    - s (str)
    Valeur de retour (bool)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> est_bien_parenthesee('())(()')
    False
    >>> est_bien_parenthesee('(()())')
    True
    >>> est_bien_parenthesee('()(')
    False
    >>> est_bien_parenthesee('(()')
    False
    """
```

Anagrammes

Pour faire cet exercice, vous aurez besoin de la fonction `supprime_caractere()` de la feuille d'exercices intermédiaires sur les chaînes.

267. Définissez une fonction `nombre_occurrences()` paramétrée par un caractère `c` et une chaîne `s` et renvoie le nombre d'occurrences de `c` dans `s`. *(Indication : utilisez une boucle* `for`*.)*

Pour savoir si deux chaînes sont des anagrammes, il suffit de compter les nombres d'occurrences du premier caractère de la première chaîne dans chacune des deux chaînes, et, si ces nombres sont égaux, de réitérer sur les deux chaînes obtenues en supprimant toutes les occurrences de ce caractère.

268. Définissez un prédicat `sont_anagrammes()` paramétré par deux chaînes et renvoyant `True` si ces deux chaînes sont anagrammes l'une de l'autre et `False` sinon, en mettant en œuvre la méthode proposée ci-dessus.

```
def sont_anagrammes(s1, s2):
    """Renvoie True ssi s1 est un anagramme de s2.
    Paramètres :
    - s1 (str)
    - s2 (str)
    Valeur de retour (bool)
    Contraintes : À COMPLÉTER !
    Exemples :
    >>> sont_anagrammes('chien', 'niche')
    True
    >>> sont_anagrammes('chien', 'nicher')
    False
    >>> sont_anagrammes('abba', 'baaa')
    False
    """
```

Exercices niveau confirmé

Formatage (épisode 1)

Dans un but de remplissages de messages ou formulaires automatiques, on souhaiterait utiliser une fonction permettant de construire une chaîne à partir d'un modèle et d'une chaîne de substitution.

```
>>> format1('{}, {}, {} !', 'Beetlejuice')
'Beetlejuice, Beetlejuice, Beetlejuice !'
>>> format1("Cher {}, vous êtes l'unique gagnant !", 'Marinette')
"Cher Marinette, vous êtes l'unique gagnant !"
```

Définissez une fonction `format1()`, paramétrée par une chaîne de modèle et une chaîne de substitution qui renvoie une chaîne obtenue en remplaçant toutes les occurences de la sous-chaîne `'{}'` par la chaîne de substitution.

Libre à vous de choisir le comportement de la fonction si la chaîne de modèle contient une accolade ouvrante `{` sans accolade fermante `}` consécutive. Ce comportement devra néanmoins être documenté.

Envisagez aussi cette situation :

```
>>> format1("Allons jusqu'au {}", "bout.") == "Allons jusqu'au bout."
True
```

Génération d'une chaîne sans doublons de lettres qui se suivent

Nous allons créer une fonction, `tirage_non_repet()`, qui génère aléatiorement une chaîne de caractères qui ne contient jamais deux lettres identiques qui se suivent. Cette fonction prendra en paramètre un entier naturel représentant la taille de la chaîne à générer.

Elle utilisera une fonction auxiliaire, `tirage_lettre()`, qui ne prend pas de paramètre et qui renvoie une lettre minuscule tirée au sort. Nous rappelons que les codes ASCII des lettres minuscules sont compris entre 97 et 122 (inclus).

Exemples d'utilisation de ces fonctions :

```
>>> tirage_lettre()
'o'
>>> tirage_lettre()
'x'
>>> tirage_non_repet(5)
'ympwp'
>>> tirage_non_repet(30)
'zmrpdknwgsqnfebxalchukpigwolnz'
```

9. Listes Python - 1re partie

Découvrir les listes Python

Coup d'œil

On découvre

- comment manipuler des collections de valeurs
- comment écrire des listes Python
- comment parcourir une liste Python ou accéder à un de ses éléments

Liste

Il est parfois nécessaire de manipuler une collection de valeurs.
Par exemple le nombre de jours de chacun des mois de l'année, ou les notes de l'ensemble des étudiants de 1re année de licence…

En Python, on utilisera par exemple les *listes*.

Ces listes Python sont des structures capables de contenir un nombre quelconque de valeurs.

Comme les chaînes de caractères ou les intervalles `range`, les listes sont des *itérables* : on va donc pouvoir réaliser une même instruction sur chacun des éléments d'une liste.

Comme les chaînes de caractères ou les intervalles `range`, les éléments des listes sont *indicés* : on va donc pouvoir accéder à un élément via son *indice* dans la liste.

Nous découvrirons ensuite d'autres caractéristiques des listes Python.

Liste — en Python

En Python, une liste de valeurs écrites entre crochets `[` et `]`, et séparées par des virgules `,` forme une liste.
On écrit par exemple

```
>>> [31, 28, 31, 30]
[31, 28, 31, 30]
>>> [12.0, 10.5, 9.5, 19., 14.5]
[12.0, 10.5, 9.5, 19.0, 14.5]
```

Le type de ces valeurs est `list`, indépendamment du type des éléments :

```
>>> type([31, 28, 31, 30])
<class 'list'>
>>> type([12.0, 10.5, 9.5, 19., 14.5])
<class 'list'>
```

Complément — Tableau et liste en informatique

En informatique, un *tableau* est une structure de données constituée d'une séquence d'éléments auxquels on peut accéder par leur indice.

En informatique, une *liste* est une structure de données qui permet de regrouper des *éléments*, et qui autorise :

- d'ajouter un élément à la liste,
- retirer un élément à la liste,

- savoir si la liste est vide.

Ces deux structures de données classiques se retrouvent dans beaucoup de langages de programmation.
Les tableaux ou listes de certains langages peuvent avoir des caractéristiques propres.

Le cas de Python est particulier. La structure de données *liste Python* est plus proche de ce que l'on nomme habituellement *tableau*.

Nous nous en tiendrons pour le moment à l'étude des listes Python.

Construire des listes

Séquence d'éléments

Une liste peut donc être écrite comme une suite entre crochets d'éléments séparés par des virgules.

Tous les types sont acceptés :

```
>>> [31, 28, 31, 30]
[31, 28, 31, 30]
>>> [12.0, 10.5, 9.5, 19., 14.5]
[12.0, 10.5, 9.5, 19.0, 14.5]
>>> ["une", "liste", "Python"]
['une', 'liste', 'Python']
>>> [True, False, False, True]
[True, False, False, True]
```

De manière plus générale que ces premiers exemples, les éléments sont donnés sous forme d'expressions qui sont évaluées :

```
>>> l = [3 * 14, 42 // 7]
>>> l
[42, 6]
>>> [3 < 14, '3' < '14', 3.14 < pi, type(pi) == float]
[True, False, True, True]
```

Une liste peut ne contenir qu'un unique élément. On écrit logiquement

```
>>> singleton = [42]
>>> singleton
[42]
```

Une liste peut ne contenir aucun élément. C'est la *liste vide*.

On écrit alors :

```
>>> []
[]
```

Concaténation et répétition

Les opérateurs maintenant habituels de concaténation – + –, et de répétition – * – peuvent être utilisés sur les listes :

```
>>> [31, 30] * 2
[31, 30, 31, 30]
>>> seq5 = ([31, 30] * 2) + [31]
>>> annee = [31, 28] + seq5 * 2
>>> annee
[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Transtypage d'itérables

Il est possible de convertir un itérable en liste en utilisant la fonction prédéfinie `list()`.
Cette fonction crée une liste dont les éléments sont ceux de l'itérable.

Nous pouvons l'utiliser sur les itérables que nous connaissons, intervalles `range` et chaînes de caractères :

```
>>> conversion_range = list(range(5, 14))
>>> conversion_range
[5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> conversion_str = list("bonjour")
>>> conversion_str
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

Accéder aux éléments d'une liste

Les éléments d'une liste sont indicés. Il est donc possible d'accéder à un élément à partir de son indice.

Comme pour les chaînes de caractères, les indices débutent à 0.
De même, la fonction prédéfinie `len()` renvoie le nombre d'éléments d'une liste, et l'indice du dernier élément d'une liste `l` est `len(l)-1`.

On accède à un élément avec la même syntaxe que pour les chaînes.

On écrit donc par exemple :

```
>>> notes = [12.0, 10.5, 9.5, 19., 14.5]
>>> notes[1]
10.5
>>> len(notes)
5
>>> notes[4]
14.5
```

Parcourir des listes

Itérer sur les éléments d'une liste

Les listes sont des *itérables*. On peut donc les parcourir à l'aide boucle **`for`**.

À chaque tour de boucle, la variable d'itération sera associée à l'élément courant de la liste.

Il est par exemple possible de calculer la somme des éléments d'une liste comme ceci :

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

Lors d'un appel `somme([6, 14, 22])`,

- lors du premier tour de boucle, `x` prend la valeur `6`,
  et le calcul de `s` produit `6`
- lors du deuxième tour de boucle, `x` prend la valeur `14`,
  et le calcul de `s` produit `20`
- lors du troisième et dernier tour de boucle, `x` prend la valeur `22`,
  et le calcul de `s` produit `42`

```
>>> des_nombres = [6, 14, 22]
>>> somme(des_nombres)
42
```

La version suivante travaille sur des chaînes de caractères :

```
def concatenation(l):
    s = ""
    for x in l:
        s += x + ' '
    return s
```

et par exemple :

```
>>> des_mots = ["une", "liste", "Python"]
>>> concatenation(des_mots)
'une liste Python '
```

Itérer sur les indices des éléments d'une liste

Les éléments des listes étant accessibles par leur indice, il est également possible de parcourir les éléments d'une liste via ces indices.
Pour parcourir l'ensemble des éléments, on fera varier les indices de `0` – inclus –, à la longue de la liste – non incluse –.

Une autre fonction pour le calcul de la somme des éléments d'une liste peut être :

```
def somme_via_indice(l):
    s = 0
    for i in range(0, len(l)):
        s += l[i]
    return s
```

On peut alors ne parcourir que certains éléments, par exemple ceux de rang pair

```
def somme_rang_pair(l):
    s = 0
    for i in range(0, len(l), 2):
        s += l[i]
    return s
```

et écrire

```
>>> des_nombres = [6, 14, 22, 1, 3, 5]
>>> 6 + 22 + 3
31
>>> somme_rang_pair(des_nombres)
31
```

ou parcourir les éléments dans l'ordre inverse :

```
def concatenation_inverse(l):
    s = ""
    for i in range(len(l)-1, -1, -1):
        s += l[i] + ' '
    return s
```

et écrire

```
>>> des_mots = ["une", "liste", "Python"]
>>> concatenation_inverse(des_mots)
'Python liste une '
```

Il est bien entendu possible de parcourir les éléments via leur indice à l'aide d'une boucle **`while`**. Cela fera l'objet d'exercices.

Memento

- les listes sont des séquences d'éléments de types quelconques
- l'écriture littérale d'une liste est une séquence d'éléments séparés par une virgule `,` encadrée de crochets `[` et `]`
- on concatène des listes avec l'opérateur `+`
- on peut répéter une liste avec l'opérateur `*`.
- les listes sont des itérables, on les parcourt avec une boucle **`for`**
- on accède à l'élément d'indice `i` d'une liste `l` avec la syntaxe `l[i]`.

Modifier une liste Python

Coup d'œil

On découvre

- que les listes Python peuvent être modifiées
- comment changer la valeur d'un élément d'une liste, ajouter ou supprimer un élément à une liste
- ce que signifient « mutable » et « mutabilité »

Mutabilité

Une liste Python peuvent être modifiée. Ce n'était pas le cas des structures rencontrées précédemment. On dit que les listes sont *mutables*.

Cette mutabilité sur une séquence de valeurs se manifeste par trois types de mutations :

- les *substitutions* ;
- les *insertions* ;
- les *suppressions*.

Substitutions

Il est possible de *substituer* la valeur d'un élément d'une liste Python par une autre. La valeur de l'élement d'indice `i` d'une liste `l` peut être remplacée par une valeur `v`. On écrit :

```
l[i] = v
```

Cette syntaxe combine :

- l'affectation =, et
- l'accès l'élément d'indice `i` de `l` : `l[i]`

```
>>> liste_cadeaux = ['pull en laine', 'console de jeux-vidéo', 'livre']
>>> liste_cadeaux[1] = 'chaussettes'
>>> liste_cadeaux
['pull en laine', 'chaussettes', 'livre']
```

Une substitution ne peut qu'utiliser un indice existant.

```
>>> liste = [49, 3]
>>> liste[2] = 404
IndexError: list assignment index out of range
```

Insertions

Une *insertion* consiste à ajouter un élément à une séquence.

Pour une liste Python `l`, on insère un élément de valeur `e` en fin de séquence avec la *méthode* `.append()`. On écrit

```
l.append(e)
```

Cette méthode ne renvoie rien, elle ne fait que muter la liste à laquelle elle est appliquée.

```
>>> a_reviser = ['for', 'chaînes', 'while']
>>> len(a_reviser)
3
>>> a_reviser.append('listes')
>>> len(a_reviser)
4
>>> a_reviser
['for', 'chaînes', 'while', 'listes']
```

On utilisera une liste Python vide et la méthode `.append` pour construire une liste Python éléments par éléments. Par exemple, dans la fonction `produit()` ci-dessous, on souhaite construire une liste Python contenant le produit élément par élément de deux listes Python passées en paramètre :

```
def produit(l1, l2):
    """Renvoie la liste contenant les produits des éléments de l1 et l2
    Paramètres :
    - l1 (list) : liste de nombres
    - l2 (list) : liste de nombres
    Valeur de retour (list) : une liste de nombres
    Contraintes : len(l1) == len(l2)
    Exemple :
    >>> produit([2, 4, 3], [1, 5, 14]) == [2, 20, 42]
    True
    """
    res = []
    for i in range(len(l1)):
        res.append(l1[i] * l2[i])
    return res
```

Suppression

Une *suppression* consiste à enlever un élément d'une séquence.

On peut supprimer n'importe quel élément d'indice `k` d'une liste Python `l` en utilisant l'instruction suivante :

```
del l[k]
```

Par exemple :

```
>>> a_faire = ['courses', 'cuisine', 'sieste', 'repas de famille', 'révisions']
>>> len(a_faire)
5
>>> del a_faire[3]
>>> len(a_faire)
4
>>> a_faire
['courses', 'cuisine', 'sieste', 'révisions']
```

Memento

- les listes Python sont mutables, c'est-à-dire modifiables
- on substitue la valeur de l'élément d'indice `i` d'une liste `l`. avec la syntaxe `l[i] = nouvelle_valeur`.
- on ajoute un élément en fin de liste avec la méthode `.append()`.
- on supprime un élément à une position donnée avec l'instruction `del`.

Exercices niveau débutant

Découvertes des listes

Quelques expressions et instructions

269. Devinez la valeur de chacune des expressions, ou l'effet de chacune des instructions ci-dessous. Vérifiez ensuite la validité de votre réponse en utilisant l'interpréteur Python.

```
l = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
l[1]
l[1][2]
len(l)
l[-1]
l[4] * 2
l[4:] * 2
l.append('samedi')
l
del l[3]
l
l[3] = 'dimanche'
l
```

Vous devriez remarquer que `l.append('samedi')`, `del l[3]` et `l[3] = 'dimanche'` ne renvoient rien.

Mutations

270. Copiez ces lignes dans un fichier source Python.

```
if __name__ == '__main__':
    ordre_croissant = [2, 4, 3, 6, 5, 1]
    perfide_ordre = ['dimanche', 'lundi', 'mardi', 'mercredi',
                     'jeudi', 'vendredi', 'samedi']

    # Mutez ces deux listes ci dessous




    # Vérification
    assert ordre_croissant == [1, 2, 3, 4, 5, 6]
    assert perfide_ordre == ['lundi', 'mardi', 'mercredi', 'jeudi',
                             'vendredi', 'samedi', 'dimanche']
    import doctest
    doctest.testmod()
```

Pour que votre code puisse être exécuté, vous devez **muter** les listes associées aux variables `ordre_croissant` et `perfide_ordre` de façon à ce que les expressions booléennes

- `ordre_croissant == [1, 2, 3, 4, 5, 6]`, et
- `perfide_ordre == ['lundi', 'mardi',…,'dimanche']`

soient toutes deux vraies.

Tant que ces deux conditions ne seront pas vérifiées, votre programme ne pourra être exécuté sans erreur…

Recherche de maximum

Copiez le code ci-dessous.

```
def maximum(l):
    """Renvoie la valeur des plus grands éléments de l.
    Paramètres :
    - l (list) : une liste de nombres
    Valeur de retour (int ou float) : la plus grande valeur dans l
    Contraintes : à compléter !
    Exemples :
    >>> maximum([4, 4, 3, 1, 2, 1, 1, 3])
    4
    """
    vmax = 0
    for e in l:
        if e > vmax:
            vmax = e
    return vmax
```

271. En utilisant l'exécution pas à pas (débogueur Thonny), complétez le tableau ci-dessous exprimant les valeurs de `e` et de `vmax` avant la boucle, puis en fin de chacune des itérations pour l'appel `maximum([1, 3, 4, 2])`.

```
# +-----+------+
# | e   | vmax |
# +-----+------+
# |     |      |  avant la boucle
# +-----+------+
# |     |      |  fin de première itération
# +-----+------+
# |     |      |  fin de deuxième itération
# +-----+------+
# |     |      |  ...
# +-----+------+
# |     |      |
# +-----+------+
```

272. Quelle est la valeur de `maximum([-1, -3])` ? Corrigez la fonction pour obtenir une valeur correcte.
273. Quelle est la valeur de `maximum([])` ? Complétez la chaîne de documentation (*docstring*) de la fonction.
274. Définissez une fonction `minimum()` prenant en paramètre une liste et renvoyant la plus petite valeur des éléments de cette liste.

Notez que les fonctions prédéfines `max()` et `min` que nous avons déjà rencontrées acceptent également un itérable - donc une liste - en paramètre dont elles renvoient le plus grand, respectivement le plus petit, élément.

Compte, somme, moyenne, voire variance

275. Définissez une fonction `nombre_occurrences()` paramétrée par une valeur `v` et une liste `l` qui renvoie le nombre d'occurrences de la valeur `v` parmi les éléments de `l`.

```
>>> nombre_occurrences(2, [4, 2, 6, 2, 2, 5, 4])
3
>>> nombre_occurrences(4, [4, 2, 6, 2, 2, 5, 4])
2
```

Notez que la méthode prédéfinie `list.count()` renvoie ce nombre d'occurences d'une valeur dans une liste.

276. Définissez une fonction `somme()` renvoyant la somme des valeurs des éléments de la liste passée en paramètre.

```
>>> somme([1, 2, 3, 4])
10
```

Notez que la fonction Python prédéfinie `sum()` renvoie cette somme des valeurs numériques d'un itérable.

277. Définissez une fonction `moyenne()` renvoyant la moyenne arithmétique des valeurs des éléments de la liste passée en paramètre.
278. La variance V d'une séquence $x_1, x_2, ..., x_n$ est définie par $V = \frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2$ où m est la moyenne de $x_1, x_2, ..., x_n$.

     Définissez une fonction `variance()` renvoyant la variance des éléments de la liste passée en paramètre.

Positivons

Pour chacune des questions suivantes, méditez sur l'usage d'une boucle `for` ou d'une boucle `while` avant d'écrire le code des fonctions.

279. Définissez un prédicat `tous_positifs()` renvoyant `True` si et seulement si la liste passée en paramètre ne contient que des éléments positifs ou nuls.
280. Définissez un prédicat `au_moins_un_positif()` renvoyant `True` si et seulement si la liste passée en paramètre contient au moins un élément positif ou nul.
281. Définissez une fonction `nombre_positifs()` renvoyant le nombre d'éléments positifs de la liste passée en paramètre.
282. Définissez une fonction `filtre_positifs()` renvoyant la liste des éléments positifs ou nuls de la liste passée en paramètre.

```
>>> filtre_positifs([0, -1, -5, 2, 8, -13])
[0, 2, 8]
```

Séquences

Séquence croissante

283. Définissez un prédicat `est_croissante()` paramétré par une liste et renvoyant `True` si et seulement si cette liste est croissante.

```
>>> est_croissante([1, 2, 4])
True
>>> est_croissante(["un", "deux", "quatre"])
False
>>> est_croissante([])
True
```

Suite arithmético-géométrique

Soit u la suite définie par récurrence.

$$\begin{cases} u_0 & = & 1 \\ u_{n+1} & = & 2u_n + 3 \end{cases}$$

284. Définissez une fonction `suite_partielle` paramétrée par un entier n et renvoyant la liste des $n+1$ premiers termes de la suite u : u_0, u_1, ..., u_{n-1} et u_n.

Exercices niveau intermédiaire

Recherche d'extrema

285. Définissez une fonction `extrema()` renvoyant une liste contenant la plus petite valeur et la plus grande valeur de la liste passée en paramètre.
   Il est possible de proposer une implantation qui ne parcourt qu'une fois la liste paramètre.

```
>>> animaux = ["cheval", "autruche", "tigre", "zèbre", "fourmi"]
>>> extrema(animaux)
['autruche', 'zèbre']
```

286. Définissez une fonction `moyenne_sans_extrema()` renvoyant la moyenne des valeurs non extrêmes d'une liste de nombres passée en paramètre.

```
>>> l = [45.0, 95.0, 95.0, 40.0, 39.0, 3.0, 44.0, 95.0, 42.0]
>>> extrema(l)
[3.0, 95.0]
>>> moyenne_sans_extrema(l) == moyenne([45.0, 40.0, 39.0, 44.0, 42.0])
True
```

287. Une version optimisée de la fonction `extrema()` peut utiliser le principe suivant pour minimiser le nombre de comparaisons :
   - soient deux éléments de la liste et un minimum et maximum actuels
   - pour trouver le minimum et maximum de ces quatre valeurs
     - comparer les deux éléments entre-eux
     - comparer le plus petit des éléments au minimum
     - comparer le plus grand des éléments au maximum
   - on a ainsi trois comparaisons plutôt que quatre

   Mettez ce principe en œuvre pour proposer une nouvelle version de `extrema()`

Autres sommes et moyennes

288. Définissez une fonction `sommes_prefixes()` renvoyant une liste contenant les sommes préfixes de la liste passée en paramètre.

```
>>> sommes_prefixes([2, 4, 3, 1]) == [2, 2+4, 2+4+3, 2+4+3+1]
True
```

289. Définissez une fonction `moyenne_ponderee()` prenant en paramètre une liste de mesures et une liste de coefficients, et renvoyant la moyenne pondérée des mesures par les coefficients.

   Par exemple, si un étudiant a obtenu respectivement les notes de 14, 15 et 11 au contrôle continu (coefficienté 2), au TP (coefficienté 1) et au devoir surveillé (coefficienté 2), alors, sa note finale pourra être calculée par `moyenne_ponderee([14.0, 15.0, 11.0], [2, 1, 2])`.

Préfixes, suffixes et facteurs

290. Définissez deux fonctions `prefixes()` et `suffixes()` renvoyant respectivement la liste des préfixes, et la liste des suffixes, de la chaîne passée en paramètre.

```
>>> prefixes('motus')
['', 'm', 'mo', 'mot', 'motu', 'motus']
>>> suffixes('fin')
['', 'n', 'in', 'fin']
```

291. Définissez une fonction `facteurs()` renvoyant la liste des facteurs (sous-chaînes) de la chaîne passée en paramètre.

```
>>> facteurs('tot')
['', 't', 'to', 'tot', 'o', 'ot', 't']
```

(Il peut y avoir des répétitions de facteurs.)

Ranges et mélanges

292. Proposez une fonction `melange(n)` qui produit une permutation des valeurs entières de `range(n)`.

Le principe suivant permet de construire une telle liste :

- à partir de la liste `a_vider` des entiers de l'intervalle `range(n)`
- construire élément par élément une liste en
  - choisissant au hasard un des éléments restant de `a_vider`
  - ajoutant cet élément à la liste en construction
  - supprimant cet élément de la liste `a_vider`
  - poursuivre tant que la liste `a_vider` n'est pas vide

293. Proposez une fonction `gather()` qui construit une liste à partir d'une liste de valeurs et une une liste d'indices sur cet exemple :

```
>>> gather(list(range(5, 12)), [2, 3, 0, 6])
[7, 8, 5, 11]
>>> gather(["William", "Jack", "Rantanplan", "Joe", "Averell"], \
           [3, 0, 1, 4])
['Joe', 'William', 'Jack', 'Averell']
```

294. Définissez une fonction `liste_melangee()` paramétrée par une liste modèle et renvoyant une nouvelle liste dont les éléments sont ceux de la liste modèle, mais dans un ordre aléatoire.
Vous utilisez par exemple les deux fonctions `melange()` et `gather` définies ci-dessus.

295. Proposez une fonction `range_a_trous(maximum, absents)` qui construit une liste de valeurs entières inférieures à `maximum` qui ne sont pas dans la liste `absents`.
Cette liste `absents` doit être triée dans l'ordre croissant.

```
>>> range_a_trous(12, [3, 6, 7, 9])
[0, 1, 2, 4, 5, 8, 10, 11]
>>> range_a_trous(18, list(range(12)))
[12, 13, 14, 15, 16, 17]
>>> range_a_trous(18, list(range(12)) + [15])
[12, 13, 14, 16, 17]
```

Zorlangue

Découpage et recollage

296. Définissez une fonction nommée `decoupage()` paramétrée par un caractère `c` et une chaîne `s`, et qui renvoie la liste des sous-chaînes de `s` découpée à chaque occurrence du caractère `c`.

```
>>> decoupage(' ', 'Il fait beau et chaud !')
['Il', 'fait', 'beau', 'et', 'chaud', '!']
>>> decoupage('@', 'prenom.nom.etu@univ-lille.fr')
['prenom.nom.etu', 'univ-lille.fr']
```

297. Définissez une fonction `recollage()` paramétrée par une chaîne `s` et une liste de chaînes, et qui renvoie une chaîne constituée des éléments de la liste recollés, avec la chaîne intercalée

```
>>> recollage(' bise ', ['Salut', 'tu', 'vas', 'bien'])
'Salut bise tu bise vas bise bien'
>>> recollage('', decoupage(' ', 'Ils viennent du bout du monde'))
'Ilsviennentduboutdumonde'
```

« Eviv Bulgroz »

298. Définissez une fonction `indices_majuscules()` paramétrée par une chaîne et renvoyant la liste des indices des majuscules dans cette chaîne.

299. Définissez une fonction `miroir_casse()` paramétrée par une chaîne `mot`, et renvoyant la chaîne constituée des caractères de `mot`, mais dans l'autre sens. De plus, les indices des lettres majuscules ne doivent pas changer. On suppose que `mot` est constituée uniquement de lettres.

```
>>> miroir_casse('Zorglub')
'Bulgroz'
>>> miroir_casse('UnivLille')
'ElliLvinu'
```

300. Définissez une fonction `zorglangue()` paramétrée par une chaîne `s` (ne contenant que des lettres et des espaces), et renvoyant cette chaîne traduite en zorlgangue (voir l'article Wikipédia `fr.wikipedia.org/wiki/Zorglangue`)

```
>>> zorlangue('Vive Zorglub')
'Eviv Bulgroz'
```

Remarquez que cette fonction peut aussi traduire la zorglangue :

```
>>> zorglangue('Esod mumixam')
'Dose maximum'
```

Exercices niveau confirmé

Indices des extrema et médiane

301. Définissez une fonction `i_extrema()` sur le même principe que la fonction `extrema()` précédemment définie, et qui renvoie la liste des indices du minimum et du maximum d'une liste donnée.

```
>>> animaux = ["cheval", "autruche", "tigre", "zèbre", "fourmi"]
>>> i_extrema(animaux)
[1, 3]
```

302. Définissez une fonction `indices_maxima()` qui renvoie la liste des indices des occurrences de la valeur maximale des éléments de la liste de nombres passée en paramètre.

```
>>> indice_maxima([2, 4, 3, 4, 1])
[1, 3]
```

303. Définissez de même une fonction `indices_minima()`.

304. Définissez une fonction `mediane()` qui renvoie la valeur médiane des éléments de la liste de nombres passée en paramètre.

     Il est possible de procéder ainsi :

     - copiez dans une variable locale la liste de nombres passée en paramètre.
     - tant qu'il y a trois nombres ou plus dans cette copie, supprimez un minimum et un maximum de cette copie.

Appliquer, filtrer, et coudre

En Python, il est possible de passer une fonction comme paramètre.

Par exemple :

```
def carre(x):
    """Le carré d'un nombre x donné"""
    return x * x
def moitie(n):
    """La moitié d'un nombre donné"""
    return n / 2

def double_appel(fonc, val):
    """Réalise deux appels imbriqués à la fonction fonc avec le paramètre val.
    Exemple :
    >>> double_appel(carre, 3)
    81
    >>> double_appel(moitie, 16)
    4.0
    """
    tmp = fonc(val)
    res = fonc(tmp)
    return res
```

305. Définissez une fonction `appliquer()` paramétrée par une fonction `fonc` et une liste `l` et qui renvoie la liste des éléments de `l` auxquels a été appliquée la fonction `fonc`.

```
>>> appliquer(carre, [1, 5, 9, -1])
[1, 25, 81, 1]
```

306. Définissez une fonction `filtrer()` paramétrée par un prédicat `p` et une liste `l` et renvoyant la liste des éléments de `l` satisfaisant `p`.

```
>>> def est_pair(n):
...     return n % 2 == 0
>>> filtrer(est_pair, [4, 4, 3, 4, 6, 1, 7])
[4, 4, 4, 6]
```

307. Définissez une fonction `coudre()` paramétrée par une fonction `f` à deux paramètres, une liste `l1` et une liste `l2` et renvoyant la liste de l'application de `f` à chaque couple d'éléments de `l1` et de `l2` de même indice.

```
>>> def addition(x, y):
...     return x+y
>>> coudre(addition, [10, 30, 40], [3, 6, 2])
[13, 36, 42]
```

Formatage et censure (épisode 2)

Formatage (épisode 2)

Dans un but de remplissages de messages ou formulaires automatiques, on souhaiterait utiliser une fonction permettant de construire une chaîne à partir d'un modèle et d'une liste de chaînes de substitution.

```
>>> format2('Bonjour {2} {1} !', ['David', 'Vincent', 'Monsieur'])
'Bonjour Monsieur Vincent !'
>>> format2('Cher {0} de {1}, qu'aimez-vous faire à {1} !', ['Marinette', 'Le Havre'])
'Cher Marinette de Le Havre, qu'aimez-vous faire à Le Havre !'
```

308. Définissez une fonction `format2()`, paramétrée par une chaîne de modèle et une liste de chaînes de substitution qui renvoie une chaîne obtenue en remplaçant toutes les occurrences des sous-chaînes de la forme `'{i}'` (avec `i` un *entier*) par les chaînes d'indice `i` de la liste.

    Il est possible d'utiliser la fonction `int()` qui, entre autres, convertit une chaîne en entier.

TODO Clarifier ! Yes :-)

Censure (épisode 2)

On souhaite pouvoir censurer une sous-chaîne dans un texte donné.

```
>>> censure2('La fonction `print()` peut être pratique.', ['input', 'print'])
'La fonction `*****()` peut être pratique.'
>>> censure1('Bal tragique à Colombey : 1 mort', list('aeiou'))
'B*l tr*g*q** à C*l*mb*y : 1 m*rt'
```

309. Définissez une fonction `censure2()`, paramétrée par chaîne et une liste de chaînes, et renvoyant le texte de la première, en remplaçant chacune des occurrences de toutes les chaînes de la seconde par autant d'étoiles `*` que nécessaire.

Exercices complémentaires

310. Proposez un traducteur en / de Leet Speak
fr.wikipedia.org/wiki/Leet_speak
311. Proposez un analyseur qui décompose les informations contenues dans une URL
fr.wikipedia.org/wiki/Uniform_Resource_Locator#Fonctionnement
312. Proposez un traducteur du Fourchelangue
https://pydefis.callicode.fr/defis/Fourchelangue/txt
313. Proposez un traducteur d'émoticônes typographiques occidentales en style nippon
https://fr.wikipedia.org/wiki/%C3%89motic%C3%B4ne#Liste_des_%C3%A9motic%C3%B4nes_typog
raphiques
314. Proposez un Jeu du pendu
fr.wikipedia.org/wiki/Le_Pendu_(jeu)
315. Proposez un programme qui filtre une séquence de lettres : les voyelles en début, les consonnes en fin
316. Proposez un programme qui filtre une séquence de caractères : les voyelles en début, les consonnes en fin, et les autres caractères au milieu

Pour chacun de ces programmes, vous veillerez à

- définir le problème
  - on pourra commencer par une version simplifiée
- décomposer le problème en sous-problèmes
- spécifier les fonctions nécessaires à la réalisation de ces sous-problèmes :
  - nom bien choisi
  - nombre et noms des paramètres, leur type
  - type de retour de la fonction
  - contraintes d'utilisation
  - docstring
  - exemples d'utilisation
- écrire le corps des fonctions
  - pour chaque fonction, on suppose que les autres fonctions existent !

Une approche progressive est préférable. Pour un problème donné

- n'hésitez pas dans un premier temps à simplifier le problème
- une fois une version mise au point pour ce problème, relâchez les contraintes qui avaient été fixées dans un premier temps
- itérer cette méthode pour aboutir à une version finale

N'oubliez pas documenter de votre approche. Expliquez en particulier quelles sont les différentes versions qui ont été développées.

10. Listes Python - 2de partie

Aliasing de listes, valeurs mutables

Coup d'œil

On découvre

- la vraie nature des variables
- la notion de référence vers une valeur
- le phénomène d'aliasing
- les conséquences de l'aliasing lors de la manipulation de listes Python qui sont mutables
- l'utilisation de Python Tutor pour visualiser l'état mémoire

Les variables sont des références

Nous avons vu dès le début de ce cours que

- une variable associe un nom à une valeur, et que
- l'opération d'*affectation* permet de lier une valeur à la variable

Nous allons approfondir cette première présentation des variables.
Ce que nous avons vu jusque maintenant n'est pas faux. C'est une approximation suffisante pour bien des situations : cela nous a permis de comprendre les programmes écrits jusque maintenant !

Et donc plus précisément nous allons découvrir que

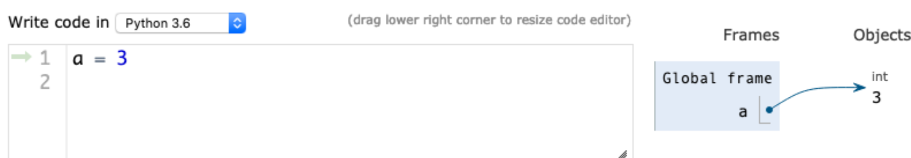
- une variable associe un nom à une *référence* à une valeur

État de la mémoire illustré par Python Tutor

L'état de la mémoire est l'ensemble des associations variable-valeur.

Nous allons utiliser des schémas générés sur le site Python Tutor `pythontutor.com/` pour illustrer cet état de la mémoire.

L'illustration ci-dessous représente l'état mémoire suite à l'exécution de l'affectation `a = 3` :



On y distingue :

- une suite d'instructions Python en partie gauche
- l'état de la mémoire en partie droite
  - lui-même divisé en deux zones *Frames* et *Objects*
  - la zone *Frames* correspond à différents blocs dans lesquels apparaîtront les variables
  - la zone *Objects* est une représentation des valeurs qui sont en mémoire

Cet état de la mémoire est celui après l'exécution de l'affectation `a = 3`.
On remarque particulièrement :

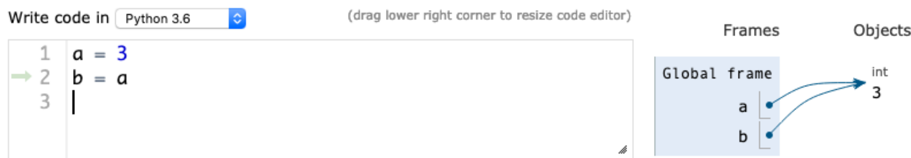
- la variable `a` qui *référence* la valeur `3`

- la flèche qui lie la variable à la valeur référencée.

Évolution de l'état de la mémoire

Quelques affectations

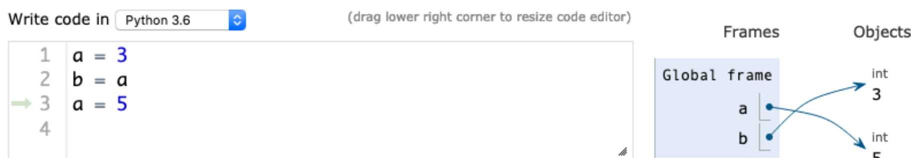
Observons l'état de la mémoire suite à l'exécution des deux affectations



Chacune des variables associe un nom et une référence à une valeur.

Il est particulièrement notable que les deux références associées à `a` et à `b` sont égales : elles désignent la même valeur `3`.

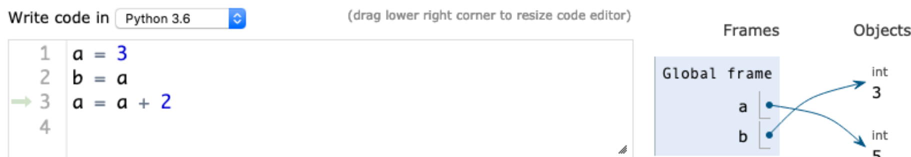
Procédons à une nouvelle affectation de `a` :



Le nom `b` reste associé à une référence qui désigne la valeur `3`.
Le nom `a` est associé à un nouvelle référence qui désigne la valeur `5`.

On obtient le même résultat avec une affectation `a = a + 2` qui est exécutée ainsi

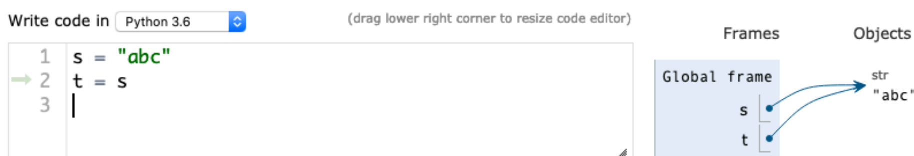
- l'expression `a + 2` est évaluée
  - elle vaut `5`
  - c'est une *nouvelle* valeur, elle apparaît comme telle
- l'affectation modifie l'association entre le nom `a` et une référence à sa valeur
  - la variable référence maintenant cette nouvelle valeur `5`‘



Avec des chaînes de caractères

Nous pouvons réaliser la même suite d'instructions avec des chaînes de caractères.

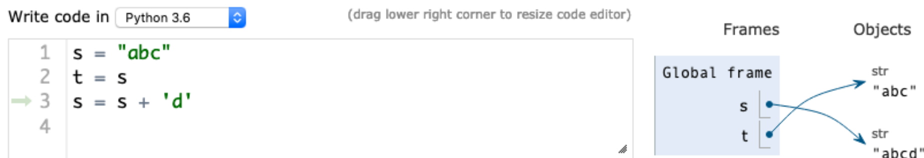
Nous allons voir apparaître des valeurs de type `str` dans la mémoire.



Les deux variables sont des références vers la même valeur `"abc"`.

Procédons à l'affectation `s = s + 'd'`. Elle est exécutée en deux étapes :

- l'expression `s + 'd'` est évaluée
  - elle vaut `"abcd"`
  - c'est une *nouvelle* valeur, elle apparaît comme telle
- l'affectation modifie l'association entre le nom `s` et une référence à sa valeur
  - la variable référence maintenant cette nouvelle valeur `"abcd`‘



Avec des listes

Nous pouvons réaliser la même suite d'instructions avec des listes.

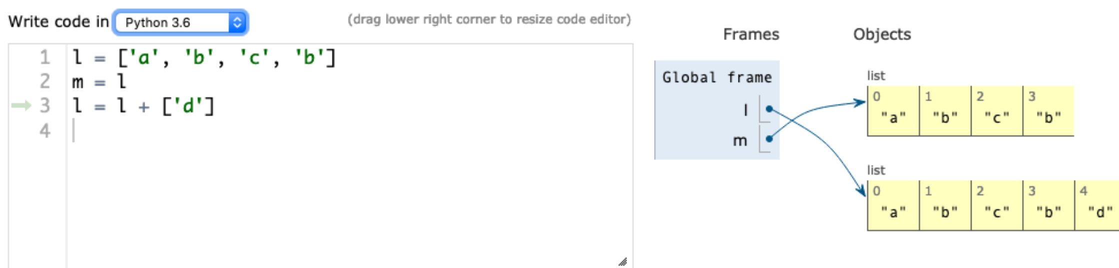
Nous allons voir apparaître des valeurs de type `list` dans la mémoire. Par exemple :

La valeur de chacun des quatre éléments de la liste `['a', 'b', 'c', 'b']` apparaît avec son indice.

Les deux variables `l` et `m` référencent cette même liste.

Comme pour les nombres ou les chaînes de caractères, une nouvelle affectation `l = l + ['d']` de la variable `l` est exécutée en deux étapes :

- l'expression `l + ['d']` est évaluée
  - elle vaut `['a', 'b', 'c', 'b', 'd']`
  - c'est une *nouvelle* valeur, elle apparaît comme telle
- l'affectation modifie l'association entre le nom `l` et une référence à sa valeur
  - la variable référence maintenant cette nouvelle valeur `['a', 'b', 'c', 'b', 'd']`‘



Une référence est un partage

Résumons.
Nous savons maintenant qu'une variable associe un nom à une *référence* vers une valeur.

Deux variables différentes peuvent donc associer deux noms à une référence vers une même valeur.
Ces deux variables partagent cette valeur.

Modifier des valeurs

Nous connaissons plusieurs types de valeurs : les nombres – `int` et `float` –, les chaînes de caractères `str`, les intervalles `range`, et les listes `list`.

Les nombres, les chaînes et les intervalles sont des valeurs qui ne peuvent être modifiées :

- on ne peut que les utiliser dans des expressions
- ces expressions produisent de *nouvelles* valeurs.

On dit que les valeurs de ces types ne sont *pas mutables*.

À l'inverse, les listes sont des valeurs qui peuvent être modifiées. Il est possible :

- de modifier un élément d'une liste
- d'ajouter un élément à une liste
- de supprimer un élément d'une liste.

On dit que les listes sont *mutables*.

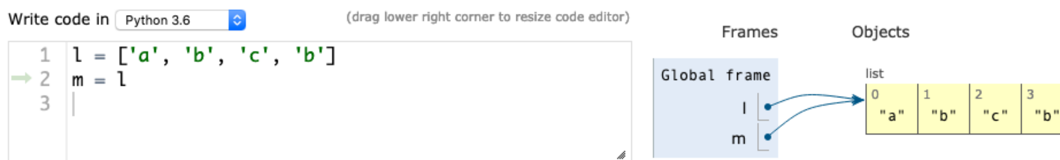
Le partage de références vers des valeurs mutables est à étudier particulièrement :
la modification de la valeur impactera l'ensemble des références vers cette valeur !

Ce phénomène est aussi appelé *aliasing* : plusieurs références permettent d'accéder à une même valeur.

Partage de listes, valeurs mutables

Deux affectations

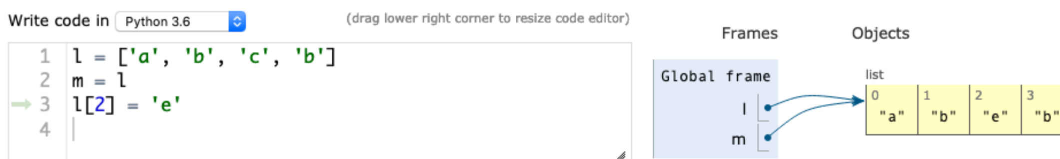
Reprenons notre exemple de manipulation de listes. Suite aux deux affectations, l'état de la mémoire est le suivant :



Les deux variables `l` et `m` référencent la même valeur `list`.

Substitution

Modifions cette valeur en substituant la valeur `'e'` à l'élément d'indice `2` de la liste. Cela est par exemple possible en écrivant `l[2]= 'e'` :



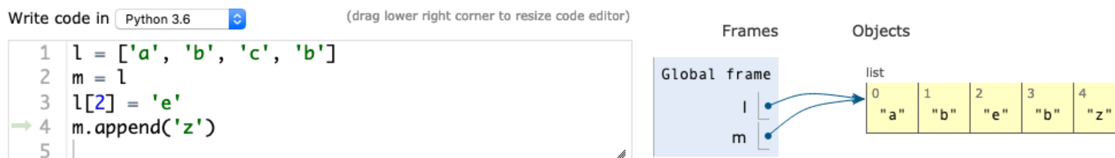
Les deux variables `l` et `m` référencent toujours cette même liste qui a été modifiée.

La valeur référencée par la variable `m` a donc été modifiée.
L'élément d'indice `2` de cette variable `m`, `m[2]`, vaut `'e'`.

Les autres moyens de modifier une valeur de type liste produiront le même effet.

Insertion

Ajoutons un élément à cette liste.
Cela peut être réalisé au choix par une opération sur `l` ou une opération sur `m`.
Choisissons d'utiliser `m('z')`.

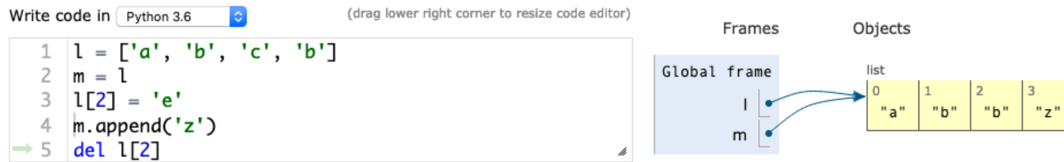


Suite à l'exécution de l'instruction `m.append('z')`, les deux variables `l` et `m` référencent toujours cette même liste qui a été modifiée.

La liste référencée par `l` est constituée de 5 éléments dont le `'z'` final.

Suppression

Supprimons maintenant un élément à cette liste.
À nouveau, cela peut être réalisé au choix par une opération sur `l` ou une opération sur `m`.
Choisissons d'utiliser `l`.



Suite à l'exécution de l'instruction `del l[2]`, les deux variables `l` et `m` référencent toujours cette même liste qui a de nouveau été modifiée.

La liste référencée par `m` est constitué de 4 éléments, le `'c'` a disparu.

Aliasing de valeurs mutables

Le phénomène d'aliasing – plusieurs références permettent d'accéder à une même valeur – doit être maitrisé quand les valeurs référencées sont mutables.
La modification de la valeur via l'une des références est partagée par l'ensemble des références à cette valeur.

Python Tutor live

Retrouvez le code présenté ici sur le site Python Tutor

- page Python Tutor `pythontutor.com/live.html#code=...`

Exécutez le code pas à pas, modifiez-le, et observez l'évolution de l'état de la mémoire.

> ## Memento
>
> - une variable associe un nom et une *référence* à une valeur
> - des variables différentes peuvent référencer une même valeur. C'est un phénomène d'*aliasing*
> - l'aliasing de valeurs mutables – les listes – doit être maîtrisé lors de la modification de telles valeurs

Copier une liste

Coup d'œil

On découvre

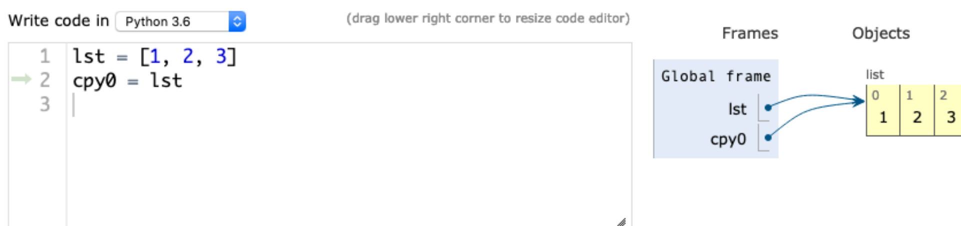
- pourquoi la copie de valeurs listes Python est indispensable
- comment réaliser une copie d'une liste Python

Nécessité de copier une liste

Les valeurs de type `list` sont mutables : les opérations de substitution d'un élément, insertion ou suppression d'un élément modifie la valeur.

On peut vouloir garder l'ancienne valeur de la liste et produire une *nouvelle* valeur par une telle modification. Par exemple ne garder dans une liste que les seuls nombres positifs, sans détruire la liste originale.

On sait qu'une simple affectation dans une variable ne produit pas une copie mais un partage de référence :



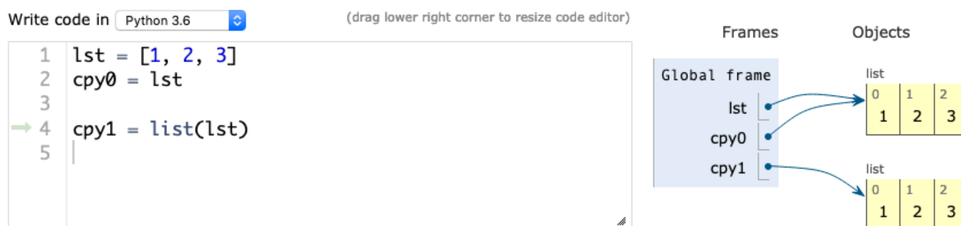
Différentes manières de copier une liste

Les manières de créer une nouvelle liste, copie d'une liste donnée, sont nombreuses.
Nous en découvrons quelques-unes ici.

Avec la fonction prédéfinie `list()`

La fonction prédéfinie `list()` construit une liste à partir d'un itérable.
Nous l'avons utilisé pour construire une liste à partir d'un intervalle `range` ou d'une chaîne de caractères.

Cette fonction permet aussi de construire une liste à partir d'une liste.
Le résultat est une nouvelle liste formée des mêmes éléments.



Par insertions successives d'éléments

Nous savons créer une liste élément par élément à partir d'une liste vide.

Nous pouvons créer une copie d'une liste en insérant un à un les éléments à l'aide de la méthode `append()` :

```python
lst = [1, 2, 3]
cpy0 = lst

cpy2 = []
for e in lst:
    cpy2.append(e)
```

En tant que tranche complète

Nous pouvons obtenir une structure formée d'une tranche des éléments d'une structure indiçable avec la syntaxe `[ imin : imax ]`.
Les bornes inférieure `imin` et supérieure `imax` peuvent être omises. Leur valeur par défaut est le plus petit, respectivement le plus grand indice de la structure.

Une copie d'une liste peut donc être créée comme une tranche du premier au dernier élément d'une liste :

```python
lst = [1, 2, 3]
cpy0 = lst

cpy3 = lst[:]
```

Par copie élément par élément

Nous savons créer une liste d'une taille donnée, par exemple par répétition d'une liste d'un élément.
La liste `[0] * n` est une liste de `n` zéros.

Nous pouvons ensuite copier un à un chacun des éléments de la liste originelle dans cette nouvelle liste à l'aide d'une boucle `for` qui parcourt les indices des listes :

```python
lst = [1, 2, 3]
cpy0 = lst

cpy4 = [0] * len(lst)
for i in range(len(lst)):
    cpy4[i] = lst[i]
```

La valeur initiale de chacun des éléments de la liste créée importe peu. Nous pouvons par exemple utiliser la valeur `None` de Python, valeur qui représente habituellement l'absence de valeur (oui..) :

```python
lst = [0, 1, 2]
cpy0 = lst

cpy4 = [None] * len(lst)
for i in range(len(lst)):
    cpy4[i] = lst[i]
```

Avec la méthode `list.copy()`

Python fournit une méthode `s.copy()` sur les valeurs de type `list` qui crée une copie de `s` :

```
lst = [1, 2, 3]
cpy0 = lst

cpy5 = lst.copy()
```

Avec la fonction `copy()` de la bibliothèque `copy`

La bibliothèque Python `copy` fournit plusieurs fonctions pour réaliser des copies de valeurs mutables.

Nous pouvons utiliser la fonction `copy()` de cette bibliothèque :

```
from copy import copy

lst = [1, 2, 3]
cpy0 = lst

cpy6 = copy(lst)
```

Python Tutor live

Retrouvez le code présenté ici sur le site Python Tutor

- page Python Tutor `pythontutor.com/live.html#code=...`

Exécutez le code pas à pas, modifiez-le, et observez l'évolution de l'état de la mémoire.

Choisir comment créer une copie d'une liste

L'ensemble des manières de copier une liste Python sont équivalentes : elles produisent toutes le même résultat ; leurs performances ne sont pas si différentes.
Il existe même d'autres manières de faire que nous n'avons pas vues.

L'utilisation des fonctions `list()` ou `copy()` est courante.

Sa syntaxe bizarre est parfois reprochée à la « copie par tranche » `lst[:]`.
Néanmoins cette manière de faire est la plus rapide.
La méthode `list.copy()` a été introduite relativement récemment dans Python pour conjuguer une meilleure lisibilité et cette efficacité : elle est implémentée de la même manière.

Copier une liste par `append()` successifs d'éléments permet aussi de ne copier que certains éléments de la liste originelle : on ajoute une instruction conditionnelle au sein de la boucle `for`.

Enfin, la fonction `copy()` de la bibliothèque `copy` a été présentée parce que cette bibliothèque fournit également d'autres fonctions que nous découvrirons plus tard.

Cela dit, faites comme il vous plaira !

Memento

- les listes sont des valeurs mutables : réaliser une copie d'une telle valeur permet de travailler sur cette copie sans modifier la liste
- des multiples manières de créer une copie sont possibles, elles sont équivalentes

Copier ou muter une liste ?

Coup d'œil

On découvre

- que deux approches sont possibles pour définir des fonctions modifiant des listes
- comment mettre en œuvre chacune des approches

Comment trier une liste Python

Soit une liste donc nous désirons trier les éléments par ordre croissant

```
l = ["un", "deux", "trois"]
```

Deux approches différentes sont possibles :

- modifier la valeur associée à `l`
- produire une nouvelle valeur, une nouvelle liste

Python fournit une fonction et une méthode correspondant à chacune des manière de faire.

Muter pour trier

Python propose une méthode applicable aux listes qui permet de trier leurs éléments.

Cette méthode modifie la liste à laquelle elle est appliquée :

```
l = ["un", "deux", "trois"]
l.sort()
```

Cette méthode ne renvoie pas de valeur.
On n'écrira donc pas `s = l.sort()` pour espérer récupérer la liste triée. L'exécution de cette instruction provoque le résultat suivant :

```
l = ["un", "deux", "trois"]
s = l.sort()
```

La variable `s` est associée à la valeur `None`, cette valeur qui indique l'absence de valeur.

Copier pour trier

Python propose aussi une fonction qui renvoie une nouvelle valeur triée :

La liste originelle n'est pas modifiée.

Muter ou copier

L'utilisation de la méthode `sort()` mute la liste. L'utilisation de la fonction `sorted()` copie la liste.

Si on n'a plus besoin de la liste originelle, la première méthode peut être employée. La seconde méthode est d'un usage plus universel.

Comment "positiver" une liste

L'exemple précédent met en évidence deux approches basées sur l'utilisation de fonctions ou méthodes prédéfinies de Python.
Voyons comment proposer nos propres fonctions pour une situation analogue.

Soit une liste de nombres, nous désirons remplacer les valeurs négatives par leur opposé.

Il est possible de proposer :

- une fonction qui modifie la liste
- une fonction qui renvoie une nouvelle liste

Muter pour positiver

Proposons une fonction `positiver()` qui accepte une liste de nombres en paramètre, et remplace chacune des valeurs négatives par leur opposée.

Cette fonction ne renverra aucune valeur. C'est une procédure. Son exécution va modifier la valeur passée en paramètre :
(*on se passe de docstring pour simplifier la présentation*)

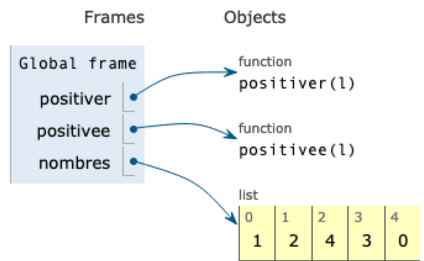
```
def positiver(l) :
    for i in range(len(l)):
        if l[i] < 0 :
            l[i] = -l[i]
```

Soit la liste `[1, -2, 4, -3, 0]`

Cette valeur est modifiée suite à un appel à `positiver()` :

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
12 nombres = [1, -2, 4, -3, 0]
13
14 positiver(nombres)
15
16
17
18
19
20
```



Copier pour positiver

Proposons une fonction `positivee()` qui accepte une liste de nombres en paramètre, et renvoie sa « *positivée* », liste dans laquelle chacune des valeurs négatives est remplacée par son opposée.

Son exécution ne va pas modifier la liste passée en paramètre.

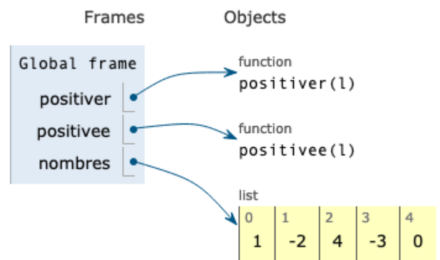
Il s'agit : 1- de copier la valeur de la liste paramètre, puis 2- de réaliser les modifications nécessaires des éléments de cette copie :
(*on se passe de docstring pour simplifier la présentation*)

```python
def positivee(l):
    res = l.copy()
    for i in range(len(res)):
        if res[i] < 0:
            res[i] = -res[i]
    return res
```

Soit à nouveau la liste `[1, -2, 4, -3, 0]`

Write code in Python 3.6 (drag lower right corner to resize code editor)

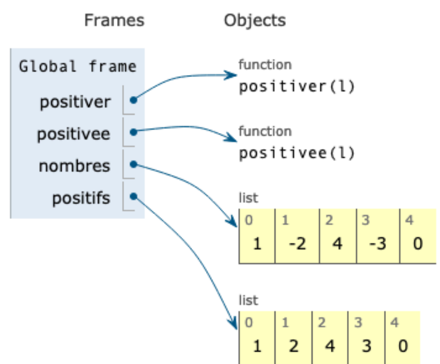
```
12 nombres = [1, -2, 4, -3, 0]
13
14
15
16
17
18
19
20
```



Suite à un appel à `positivee()`, une liste résultat est créée et la valeur originelle non modifiée :

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
12 nombres = [1, -2, 4, -3, 0]
13
14 positifs = positivee(nombres)
15
16
17
18
19
20
21
22
23
24
```



Muter ou copier : deux approches

Deux approches sont donc utilisées pour obtenir une liste modifiée à partir d'une valeur originelle :

1. proposer une procédure
    - qui modifie la valeur passée en paramètre
    - et ne renvoie rien

On parle de « *modification en place* », ou d'« *effet de bords* ».

On parle aussi d'approche *procédurale* : l'exécution d'une procédure modifie l'état de la mémoire, sans renvoyer de valeur.

2. proposer une fonction

   - qui renvoie une nouvelle liste, modification du paramètre.

   On parle d'approche *fonctionnelle* : les valeurs ne sont pas modifiées, mais de nouvelles valeurs sont produites.

Il est important de bien préciser – par exemple dans la docstring – quelle approche est choisie quand on développe une fonction travaillant sur des listes.
Il en sera de même pour les autres valeurs mutables que nous rencontrerons plus tard.

Ces deux approches fondent également des paradigmes de programmation : on parle de programmation procédurale ou impérative d'un côté, de programmation fonctionnelle de l'autre.

Memento

- l'approche dite procédurale modifie les valeurs de ses paramètres en faisant usage de mutations de listes
- l'approche dite fonctionnelle renvoie de nouvelles valeurs en faisant usage de copies de listes
- il est essentiel de bien documenter l'approche utilisée

Exercices niveau débutant

Suivre les alias

Soit la suite d'instructions suivantes :

```
a = 1
b = [a, 2*a]
c = b
d = b + [a]
b.append(4)
a = 42
b[1]= 42
e = c[1] == d[1]

c = d
d[1] = 41
f = c[1] == d[1]

b = c + d
b[2] = 14
g = b[2] == c[2]

del c[1]
h = d[0] == d[1]
```

317. Dessiner les valeurs associées à chacune des variables à la manière des schémas Python Tutor.
Vérifiez ensuite votre proposition via une exécution pas à pas sur la page Python Tutor `pythontutor.com/live.html#co`

Retour sur quelques fonctions

Dans cet exercice, faites bien attention à écrire des procédures qui **mutent** les listes passées en paramètres et ne renvoient rien.

318. Définissez une procédure `reflechit()` qui mute la liste passée en paramètre de façon à ce que les éléments de cette liste soient dans l'ordre inverse.

```
>>> l = [2, 5, 4, 3]
>>> reflechit(l)
>>> l
[3, 4, 5, 2]
```

319. Définissez une procédure `filtre_positifs()` qui supprime de la liste passée en paramètre tous les entiers strictement négatifs.

```
>>> l = [2, -4, 3, 0, 5, -1]
>>> filtre_positifs(l)
>>> l
[2, 3, 0, 5]
```

320. Définissez une procédure `cumule()` qui mute les éléments d'une liste de nombre pour que celle-ci devienne la somme cumulée de l'originale.

```
>>> l = [2, 4, 5, 1]
>>> cumule(l)
>>> l == [2, 2+4, 2+4+5, 2+4+5+1]
True
```

321. Définissez une procédure `melange()` qui permute aléatoirement les éléments d'une liste de longueur n. Cette fois-ci, on propose, de tirer n fois un indice au hasard dans l'intervalle $\{0, 1, 2, ..., n-1\}$, de supprimer de la liste l'élément associé à cet indice, et de l'ajouter en fin de liste.

```
>>> l = [1, 2, 3, 4]
>>> melange(l)
>>> l
[1, 3, 4, 2]
```

Insérer une valeur

Insérer à une position donnée

Il s'agit d'insérer une valeur dans une liste à une position donnée, et pas nécessairement en bout de liste.

Ainsi, insérer la valeur `'loup'` en position `2` de la liste `['chou', 'chèvre', 'bâton', 'feu']` produira `['chou', 'chèvre', 'loup', 'bâton', 'feu']`.

La position d'insertion est une valeur comprise entre

- `0` : la valeur doit être insérée en début de liste
- la longueur de la liste : la valeur doit être insérée en fin de liste

322. Proposez deux variantes pour insérer une valeur dans une liste :

    - une procédure dans une approche procédurale,
    - une fonction dans une approche fonctionnelle.

    Portez une attention particulière aux noms que vous donnerez à cette procédure et cette fonction.

Insérer dans une liste croissante

Il s'agit maintenant d'insérer une valeur dans une liste ordonnée croissante.

La valeur sera insérée à une position telle que cette propriété de la liste soit conservée.

323. Proposez éventuellement deux variantes pour insérer une valeur dans une liste ordonnée croissante :

    - une procédure dans une approche procédurale,
    - une fonction dans une approche fonctionnelle.

    Précisez la ou les approches choisies.

Tri par insertion

Soit une liste de valeurs à trier.

Il est possible de procéder par insertions successives des valeurs de la liste dans la liste triée.

Ces insertions se font à l'aide des fonctions/procédures définies à l'exercice précédent.

324. Proposez, au choix, une procédure ou une fonction pour réaliser un tri par insertion d'une liste de valeurs.

Manipuler des piles

En informatique, une pile est une structure de données qui propose trois opérations primitives :

- *empiler* qui ajoute un élément au sommet de la pile
- *dépiler* qui supprime l'élément en sommet de pile et renvoie cette valeur
- *vide* qui vérifie si la pile est vide.

Aucune autre opération n'est permise.

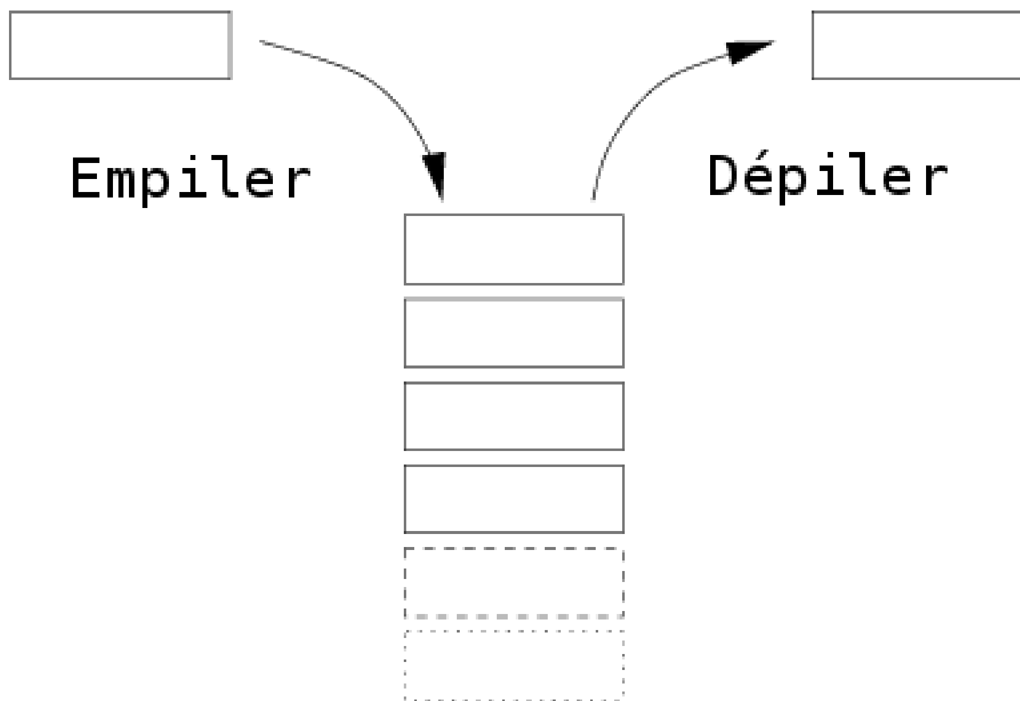


Image Wikimedia Commons
The original uploader was 16@r at French Wikipedia., CC BY-SA 3.0
commons.wikimedia.org/wiki/File:PrimitivesPile.png

Ainsi une pile fonctionne selon le principe du *dernier entré, premier sorti*, comme une pile d'assiettes.

Interface procédurale de manipulation de piles

Nous définissons une interface procédurale pour manipuler des piles à base de quatre primitives :

```
def empiler(v, p):
    """
    Ajoute la valeur v en sommet de la pile p.
    Effet de bords : la valeur p est modifiée
    """

def depiler(p):
    """
    Supprime la valeur en sommet de la pile p.
    Renvoie cette valeur
    Contraintes : la pile doit être non vide.
    Effet de bords : la valeur p est modifiée
    """

def est_vide(p):
    """
    Vrai ssi la pile est vide.
    """

def pile_vide():
    """
    La pile vide.
    """
```

Implantation de piles

Il est possible d'utiliser une liste Python pour mettre en œuvre une pile.
On choisit par exemple de mémoriser

- la valeur en fond de pile dans le 1er élément de la liste
- la valeur en sommet de pile dans le dernier élément de la liste.

Ainsi,

- empiler une valeur revient à insérer un cette valeur en fin de liste,
- dépiler revient à supprimer le dernier élément de la liste.

Ce choix n'est connu que de la personne qui propose une implémentation des piles.
Il n'est pas connu de la personne qui va utiliser les fonctions.

325. Proposez une implémentation des primitives `empiler()`, `depiler()`.

La pile vide est une liste vide. Et donc

- la fonction `pile_vide()` renvoit simplement cette valeur,
- le prédicat `est_vide` compare la valeur de son paramètre avec cette valeur.

326. Proposez une implémentation des primitives `pile_vide()` et `est_vide()`.

Des listes aux piles, et retour

Définissons deux fonctions utilitaires pour passer d'une liste à une pile, et inversement :

```
def pile_de_liste(l):
    """
    Une pile formée des éléments de la liste l.
    La valeur l[0] en sommet de pile, la valeur l[-1] en fond de pile.
    """

def liste_de_pile(p):
    """
    Vide la pile dans une liste l
    Le fond de pile est en l[0], le sommet en fin de liste.
    Retourne l.
    Effet de bord : p est vide à l'issue de la fonction.
    """
```

Ces deux fonctions utilitaires peuvent être écrites sans connaître la manière dont les piles sont implémentées. Elle ne doivent utiliser que les seules primitives de l'interface définie sur les piles : `empiler()`, `depiler()`, `est_vide()`, et `pile_vide()`.

327. Proposez une définition des deux fonctions `pile_de_liste()` et `liste_de_pile(p):`.

Piles de triage

À l'image des gares de triage nous allons utiliser des piles pour réarranger les éléments d'une liste.

Soit une liste d'entiers à réarranger de telle sorte que les entiers pairs soient au début de la liste, et les entiers impairs en fin de liste.
Il est possible de procéder ainsi :

- mettre les éléments dans une pile
- vider la pile dans deux piles, une pile des entiers pairs, et une pile des entiers impairs
- vider la pile des entiers pairs dans une nouvelle pile
- vider la pile des entiers impairs dans cette même pile

328. Proposez une fonction `triage()` qui accepte une liste d'entiers en paramètre et renvoie une nouvelle liste contenant les valeurs paires puis les valeurs impaires.

     On suivra le principe présenté.
     On utilisera aucune autre fonction que les primitives définies sur les piles et les deux fonctions auxiliaires de transformation depuis et vers des listes.

Bon parenthésage

Nous avons déjà traité du bon parenthésage d'expressions. Nous n'avons considéré qu'un unique type de parenthèse.
Il s'agit ici de vérifier le bon parenthésage d'expressions pouvant comporter plusieurs types de parenthèses : `(` et `)`, `[` et `]`, `{` et `}`.

Une expression est correctement parenthésée si chaque parenthèse fermante coïncide avec la parenthèse ouvrante correspondante.

Soit une expression donnée sous la forme d'une chaîne de caractères. Un algorithme pour vérifier que l'expression est bien parenthésé est le suivant. Il utilise une pile de caractères :

- itérer sur les caractères de la chaîne
- ignorer les caractères qui ne sont pas des parenthèses
- une parenthèse ouvrante est empilée
- pour une parenthèse fermante :
  - récupérer le caractère en sommet de pile
  - poursuivre s'il s'agit de la parenthèse ouvrante correspondante
  - sinon, l'expression n'est pas bien parenthèse
- à l'issue des itérations
  - l'expression est bien parenthésée si la pile est vide

329. Proposez un prédicat `correspondent()` :

```
def correspondent(ouvrante, fermante):
    """
    Vrai ssi le caractère ouvrante et le caractère fermante sont deux
    parenthèses qui correspondent.
    Paramètres :
    - ouvrante : caractère
    - fermante : caractère
    Valeur de retour : booléen
    Contraintes : ouvrante parmi ( { [, fermante parmi ) } ]
    Exemples :
    >>> correspondent('{', '}')
    True
    >>> correspondent('[', ')')
    False
    >>> correspondent(')', '(')
    False
    """
```

330. Proposez unn prédicat `bien_parenthesee()` qui vérifie qu'une expression fournie en paramètre sous la forme d'une chaîne de caractères est bien parenthésée.

```
>>> bien_parenthesee('(2+3) - [(a+b).(a-b)]')
True
>>> bien_parenthesee('( ... ( ] [ ) )')
False
>>> bien_parenthesee('[ ... { ... } ...')
False
>>> bien_parenthesee('')
True
```