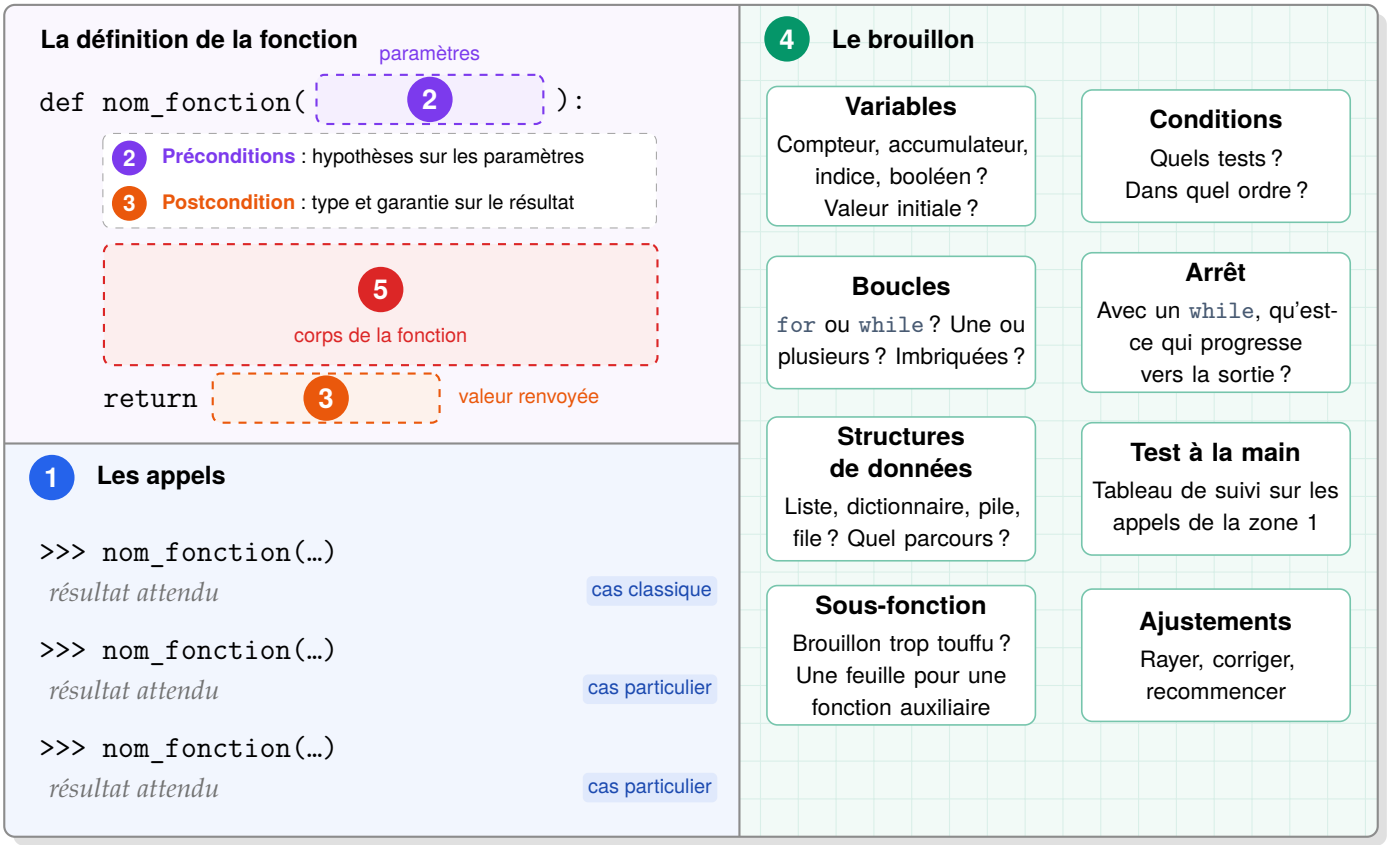
Une méthode pour écrire une fonction

Quand on doit écrire une fonction, il est tentant de taper du code tout de suite, puis de le modifier au hasard jusqu'à ce que le programme semble fonctionner. On passe alors beaucoup de temps à corriger des erreurs sans avoir compris ce que l'on cherchait.

Il est plus efficace de répondre d'abord, sur une feuille, à deux questions dans cet ordre : que doit faire la fonction, puis comment le fait-elle. L'ordinateur sert ensuite à vérifier une solution déjà trouvée, et non à la chercher.

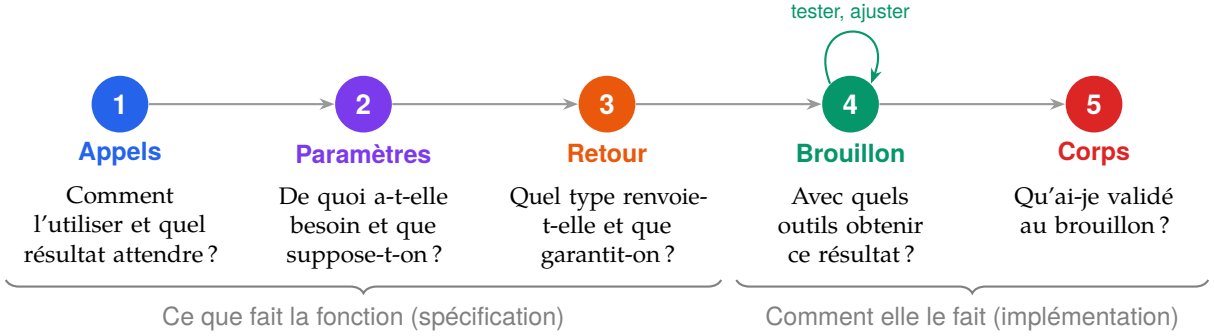
① Les trois zones

La colonne de gauche contient deux zones : en haut, la définition de la fonction, à compléter ; en bas, les appels de la fonction. La colonne de droite forme la troisième zone et sert de brouillon.



② Les cinq étapes

Les numéros ne suivent pas l'ordre de lecture du programme : le corps de la fonction est écrit en dernier. Les trois premières étapes décrivent ce que fait la fonction (sa spécification) ; les deux dernières, comment elle le fait (son implémentation).



Pourquoi commencer par les appels ? Si l'on n'arrive pas à prévoir à la main le résultat d'un appel, c'est que l'énoncé n'est pas encore compris : il est inutile d'aller plus loin. Ces appels serviront ensuite de tests.

La docstring, contrat de la fonction. Avant toute recherche, on y écrit :

— les préconditions, c'est-à-dire ce que l'on suppose sur les paramètres ;

— la postcondition, c'est-à-dire le type du résultat et ce que l'on garantit sur lui.

La postcondition est placée en tête de la fonction et non près du `return`, car la docstring est ce que lit la personne qui utilise la fonction (avec `help(nom_fonction)`), sans regarder le code.

Les fonctions qui ne renvoient rien. Une fonction qui trie une liste sur place ou qui empile un élément renvoie `None`. Sa postcondition décrit alors l'état de la structure après l'appel, et c'est cet état que vérifient les tests de la zone 1.

3 Trouver les cas particuliers

Après le cas classique, on passe en revue les situations suivantes et l'on garde celles qui ont un sens pour la fonction : une structure vide, un seul élément, l'élément en tête, l'élément en fin, un élément absent, des doublons, des valeurs nulles ou négatives, une valeur limite, etc.

4 Le brouillon

Dans la zone 4, on liste ce qui pourrait servir à écrire le corps :

— les variables et leur valeur initiale ;

— les conditions et l'ordre dans lequel les tester ;

— le type de boucle : `for` si le nombre de tours est connu à l'avance, `while` s'il dépend d'une condition ;

— l'organisation des boucles, successives ou imbriquées ;

— la structure de données adaptée (liste, tuple, dictionnaire, pile, file, etc.) et la meilleure façon de la parcourir, par exemple par valeur ou par indice pour une liste ou un tuple, par clé, par valeur (avec `d.values()`) ou par couple clé-valeur (avec `d.items()`) pour un dictionnaire ;

— un test à la main sur les appels de la zone 1, dans un tableau de suivi des variables.

Une boucle `while` doit s'arrêter : on cherche un variant de boucle, c'est-à-dire une quantité entière positive qui diminue strictement à chaque tour. Si le brouillon devient touffu, une partie du travail mérite sans doute sa propre fonction : on la cherche sur une nouvelle feuille, puis on l'utilise comme un outil.

Le brouillon n'a pas à être propre : c'est la seule zone où l'erreur est attendue. Une piste rayée n'est pas du temps perdu, elle montre ce qui ne fonctionne pas.

5 Deux exemples

5.1 Compter les occurrences d'un caractère

On cherche une fonction `nb_occurrences` qui renvoie le nombre de fois où le caractère `car` apparaît dans la chaîne `mot`. Le nombre de tours est connu : une boucle `for` convient.

La définition de la fonction (2)

```
def nb_occurrences(car, mot):
```

> **Pré** : `mot` est une chaîne de caractères et `car` un caractère.
> **Post** : renvoie un entier, le nombre de fois où `car` apparaît dans `mot`.

(5)

```
    compteur = 0
    for c in mot:
        if c == car:
            compteur = compteur + 1
```

```
    return compteur
```
(3)

(1) Les appels

```
>>> nb_occurrences("a", "banane")
2
```
cas classique

```
>>> nb_occurrences("z", "banane")
0
```
caractère absent

```
>>> nb_occurrences("a", "")
0
```
chaîne vide

(4) Le brouillon

Variable : un compteur, initialisé à 0.

Boucle : il faut regarder tous les caractères de `mot`.

~~`while i < len(mot):`~~
Un `for` suffit : le nombre de tours est connu.

`for c in mot:`

Condition : si `c == car`, le compteur augmente de 1.

Arrêt : un `for` s'arrête toujours à la fin de la chaîne.

Test à la main : sur l'appel classique, le compteur prend successivement les valeurs 0, 1, 1, 2, 2 et 2 ; on obtient bien 2.

5.2 Chercher l'indice d'un élément

On cherche une fonction `indice` qui renvoie l'indice de la première occurrence de `x` dans `elements`, ou -1 si `x` est absent. Il faut s'arrêter dès que l'élément est trouvé : le nombre de tours n'est pas connu à l'avance.

La définition de la fonction (2)

```
def indice(x, elements):
```

> **Pré** : `elements` est une liste.
> **Post** : renvoie un entier, l'indice de la première occurrence de `x` dans `elements`, ou -1 si `x` est absent.

(5)

```
    i = 0
    n = len(elements)
    while i < n and elements[i] != x:
        i = i + 1
    if i == n:
        i = -1
```

```
    return i
```
(3)

(1) Les appels

```
>>> indice(5, [2, 5, 7, 5])
1
```
cas classique

```
>>> indice(2, [2, 5])
0
```
élément en tête

```
>>> indice(9, [2, 5])
-1
```
élément absent

```
>>> indice(3, [])
-1
```
structure vide

(4) Le brouillon

Variables : un indice `i`, initialisé à 0, et la longueur `n` de la liste.

Boucle : s'arrêter dès que `x` est trouvé.

~~`for e in elements:`~~
Un `for` parcourt toute la liste : il faut un `while`.

Condition pour continuer : `i < n` et `elements[i] != x`.
L'ordre compte : si `i` vaut `n`, `elements[i]` provoquerait une erreur.

Arrêt : `n - i` est un entier positif qui diminue de 1 à chaque tour ; c'est un variant.

Après la boucle : si `i == n`, `x` est absent et l'on renvoie -1.

Test à la main : tableau de suivi ci-dessous.

Voici le tableau de suivi pour l'appel `indice(9, [2, 5])`, qui teste le cas de l'élément absent :

`i`	`i < n`	`elements[i] != x`	Suite
0	`True`	`True`	on continue
1	`True`	`True`	on continue
2	`False`	non évalué	on sort

En sortie, `i` vaut 2, c'est-à-dire `n` : `x` est absent et la fonction renvoie −1, comme prévu. Python n'évalue pas `elements[2]`, car la première condition est déjà fausse.

6 Sur ordinateur

On recopie la fonction avec sa docstring et l'on transforme chaque appel de la zone 1 en assertion. Si une assertion échoue, on lit d'abord le message d'erreur et l'appel concerné : une faute de frappe ou un oubli se corrige directement. En revanche, si l'on ne comprend pas l'erreur ou si l'on modifie le code sans savoir pourquoi, il vaut mieux revenir au brouillon.

```
def indice(x, elements):
    """Pré : elements est une liste.
    Post : renvoie un entier, l'indice de la première occurrence
    de x dans elements, ou -1 si x est absent."""
    i = 0
    n = len(elements)
    while i < n and elements[i] != x:
        i = i + 1
    if i == n:
        i = -1
    return i

assert indice(5, [2, 5, 7, 5]) == 1
assert indice(2, [2, 5]) == 0
assert indice(9, [2, 5]) == -1
assert indice(3, []) == -1
```

À éviter	À privilégier
Ouvrir l'éditeur dès la lecture de l'énoncé.	Écrire d'abord les appels et les résultats attendus.
Oublier les cas particuliers.	Passer en revue la liste des cas particuliers.
Corriger au hasard jusqu'à ce que « ça marche ».	Dérouler l'algorithme à la main et ajuster le brouillon.
Tester sur un seul exemple, dans la console.	Transformer tous les appels en assertions.

7 S'entraîner

7.1 Somme des chiffres d'un entier

On cherche une fonction `somme_chiffres` qui renvoie la somme des chiffres d'un entier positif ou nul, par exemple 4 + 0 + 7 + 1 = 12 pour 4 071. On rappelle que `n % 10` donne le dernier chiffre de `n` et que `n // 10` le supprime. La spécification est donnée ; il reste à trouver le corps et la valeur renvoyée.

La définition de la fonction

```
def somme_chiffres(n):
    """Pré : n est un entier positif ou nul.
       Post : renvoie un entier, la somme des chiffres de n."""
    # 5
    return # 3
```

1 Les appels

```
>>> somme_chiffres(4071)
12                                  cas classique
>>> somme_chiffres(7)
7                                   un seul chiffre
>>> somme_chiffres(100)
1                                   des zéros
>>> somme_chiffres(0)
0                                   nombre nul
```

4 Le brouillon

7.2 Nombre d'élèves admis

On cherche une fonction `nb_admis` qui compte les élèves ayant au moins 10 sur 20 dans un dictionnaire de notes. Il faudra se demander quel parcours du dictionnaire est le plus adapté.

La définition de la fonction

```
def nb_admis(notes):
    """Pré : notes est un dictionnaire qui associe à chaque nom
       une note sur 20.
       Post : renvoie un entier, le nombre d'élèves dont la note
       est supérieure ou égale à 10."""
    # 5
    return # 3
```

1 Les appels

```
>>> nb_admis({"Ana": 12, "Léo": 8})
1                                   cas classique
>>> nb_admis({"Zoé": 10})
1                                   valeur limite
>>> nb_admis({})
0                                   dictionnaire vide
```

4 Le brouillon

⑧ Feuille de recherche à imprimer

Fonction à écrire : .....

La définition de la fonction

```
def ..... ( ) :
```

Pré : .....

Post : .....

.....

```
return
```

1 Les appels

cas classique

>>> .....

résultat : .....

cas particulier

>>> .....

résultat : .....

cas particulier

>>> .....

résultat : .....

cas particulier

>>> .....

résultat : .....

4 Le brouillon

Tableau de suivi

Penser à : variables et valeurs initiales, conditions, boucles (`for` ou `while`), arrêt des boucles `while`, structures de données et leur parcours, fonction auxiliaire si le brouillon devient touffu.