IFT-1004 - Introduction à la programmation

Module 3 : Fonctions et décomposition fonctionnelle

Honoré Hounwanou, ing.

Département d'informatique et de génie logiciel
Université Laval

Table des matières

Les fonctions : déclaration et utilisation

Les fonctions : déclaration et utilisation

Objectif

- Utiliser la décomposition fonctionnelle comme voie de simplification d'un problème et de structuration d'une solution

Les fonctions : déclaration et utilisation

Objectif

- Utiliser la décomposition fonctionnelle comme voie de simplification d'un problème et de structuration d'une solution
- Pouvoir déclarer et utiliser des fonctions en Python

Les fonctions : déclaration et utilisation

Objectif

- Utiliser la décomposition fonctionnelle comme voie de simplification d'un problème et de structuration d'une solution
- Pouvoir déclarer et utiliser des fonctions en Python
- Maîtriser le passage de valeurs en paramètres

La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement

La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement
- Par exemple : on doit déterminer si des nombres sont premiers

La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement
- Par exemple : on doit déterminer si des nombres sont premiers
- *n* est premier s'il n'a comme diviseurs que 1 et lui-même, *n*

La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement
- Par exemple : on doit déterminer si des nombres sont premiers
- *n* est premier s'il n'a comme diviseurs que 1 et lui-même, *n*
- Sans plus d'information, on peut utiliser une *fonction* `premier(n)`, en supposant que la valeur *retournée* par celle-ci sera une valeur booléenne :

La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement
- Par exemple : on doit déterminer si des nombres sont premiers
- *n* est premier s'il n'a comme diviseurs que 1 et lui-même, *n*
- Sans plus d'information, on peut utiliser une *fonction* `premier(n)`, en supposant que la valeur *retournée* par celle-ci sera une valeur booléenne :

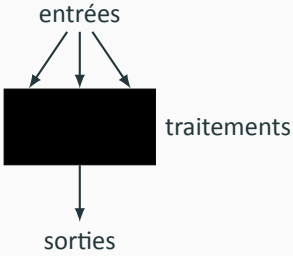
La décomposition fonctionnelle

- Objectif : imaginer une solution en termes de *boîtes noires* que l'on décrira ultérieurement → façon de concevoir des solutions progressivement
- Par exemple : on doit déterminer si des nombres sont premiers
- *n* est premier s'il n'a comme diviseurs que 1 et lui-même, *n*
- Sans plus d'information, on peut utiliser une *fonction* `premier(n)`, en supposant que la valeur *retournée* par celle-ci sera une valeur booléenne :

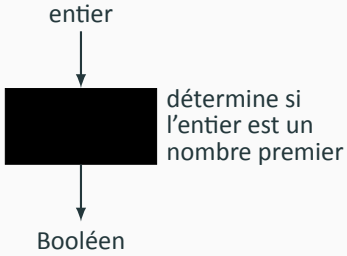
```
if premier(4):
    ...
else:
    ...
```

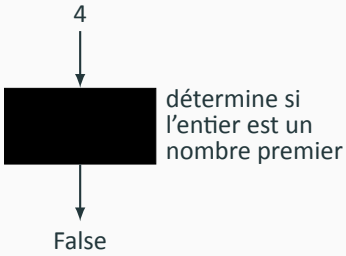
Une boîte noire ?

On voit une fonction comme une *boîte noire*.



La fonction `premier(arg)`





La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même

La décomposition fonctionnelle

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*
- On veut faire **abstraction** de ce détail dans notre code, tout en le rendant plus facile à lire. On va donc créer une fonction `diviseur(a, b)` retournant un Booléen, tel que :

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*
- On veut faire **abstraction** de ce détail dans notre code, tout en le rendant plus facile à lire. On va donc créer une fonction `diviseur(a, b)` retournant un Booléen, tel que :
  - `diviseur(a, b) == True` si `a % b == 0`

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*
- On veut faire **abstraction** de ce détail dans notre code, tout en le rendant plus facile à lire. On va donc créer une fonction `diviseur(a, b)` retournant un Booléen, tel que :
  - `diviseur(a, b) == True` si `a % b == 0`
  - `diviseur(a, b) == False` autrement

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*
- On veut faire **abstraction** de ce détail dans notre code, tout en le rendant plus facile à lire. On va donc créer une fonction `diviseur(a, b)` retournant un Booléen, tel que :
  - `diviseur(a, b) == True` si `a % b == 0`
  - `diviseur(a, b) == False` autrement
- On peut utiliser `diviseur(a, b)` dans notre code de la fonction `premier(n)`, sans élaborer `diviseur(a, b)`

La décomposition fonctionnelle

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même
- *n* n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- Si `n % i == 0`, alors *n* se divise exactement par *i*
- On veut faire **abstraction** de ce détail dans notre code, tout en le rendant plus facile à lire. On va donc créer une fonction `diviseur(a, b)` retournant un Booléen, tel que :
  - `diviseur(a, b) == True` si `a % b == 0`
  - `diviseur(a, b) == False` autrement
- On peut utiliser `diviseur(a, b)` dans notre code de la fonction `premier(n)`, sans élaborer `diviseur(a, b)`
- Les noms de variables dans la définition d'une fonction sont appelés **paramètres**.

La fonction `diviseur(arg, arg)`

- `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`

La fonction `diviseur(arg, arg)`

- `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`
- La position des valeurs à l'appel de la fonction est importante

La fonction `diviseur(arg, arg)`

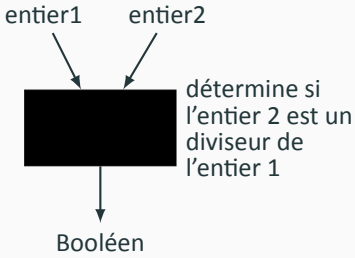
- `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`
- La position des valeurs à l'appel de la fonction est importante
  - `diviseur(12, 4)` n'est pas la même chose que `diviseur(4, 12)`

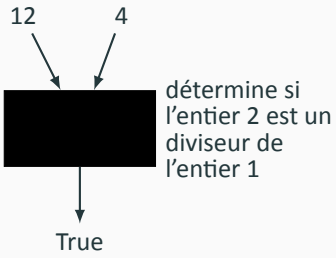
La fonction `diviseur(arg, arg)`

- `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`
- La position des valeurs à l'appel de la fonction est importante
  - `diviseur(12, 4)` n'est pas la même chose que `diviseur(4, 12)`

La fonction `diviseur(arg, arg)`

- `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`
- La position des valeurs à l'appel de la fonction est importante
  - `diviseur(12, 4)` n'est pas la même chose que `diviseur(4, 12)`





La fonction `premier(n)`

- *n* est premier s'il n'a comme diviseurs que 1 et lui-même

La fonction `premier(n)`

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$

La fonction `premier(n)`

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` sinon.

La fonction `premier(n)`

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` sinon.

La fonction `premier(n)`

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` sinon.

```
# On suppose que n >= 1

prem = True  # n est supposé premier
             # jusqu'à preuve du contraire

...

# prem contient la réponse
```

La fonction `premier(n)`

- n est premier s'il n'a comme diviseurs que 1 et lui-même
- n n'est pas premier s'il a d'autres diviseurs, compris entre 2 et $n - 1$
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` sinon.

```
# On suppose que n >= 1

prem = True  # n est supposé premier
             # jusqu'à preuve du contraire

nb = 2  # On "balaie" les entiers entre 2 et n-1
while nb <= n - 1 and prem == True:
    prem = not diviseur(n, nb)
    nb = nb + 1

# prem contient la réponse
```

La fonction `premier(n)`

- Lorsqu'un vérifie le contenu d'une variable Booléenne dans une expression conditionnelle, on peut omettre le `== True`

La fonction `premier(n)`

- Lorsqu'un vérifie le contenu d'une variable Booléenne dans une expression conditionnelle, on peut omettre le `== True`

La fonction `premier(n)`

- Lorsqu'un vérifie le contenu d'une variable Booléenne dans une expression conditionnelle, on peut omettre le `== True`

```
# On suppose que n >= 1

prem = True  # n est supposé premier
             # jusqu'à preuve du contraire

nb = 2  # On "balaie" les entiers entre 2 et n-1
while nb <= n - 1 and prem:
    prem = not diviseur(n, nb)
    nb = nb + 1

# prem contient la réponse
```

Déclaration d'une fonction en Python

```
def premier(n):
    # On suppose que n >= 1

    prem = True  # n est supposé premier
                 # jusqu'à preuve du contraire

    nb = 2  # On "balaie" les entiers entre 2 et n-1
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    # prem contient la réponse
    return prem
```

Utilisation d'une fonction en Python

```
if premier(4):
    print("Le nombre 4 est premier!?")
else:
    print("Le nombre 4 n'est pas premier!")

x = 7
if premier(x):
    print("Le nombre", x, "est premier!")
else:
    print("Le nombre", x, "n'est pas premier!")
```

La fonction `diviseur(a, b)`

- b est un diviseur de a si le reste de la division de a par b est 0

La fonction `diviseur(a, b)`

- b est un diviseur de a si le reste de la division de a par b est 0
- Si `a % b == 0`, alors b est un diviseur de a

La fonction `diviseur(a, b)`

- b est un diviseur de a si le reste de la division de a par b est 0
- Si `a % b == 0`, alors b est un diviseur de a
- La fonction `diviseur(a, b)` retourne `True` si b est un diviseur de a, et `False` autrement.

La fonction `diviseur(a, b)`

- `b` est un diviseur de `a` si le reste de la division de `a` par `b` est `0`
- Si `a % b == 0`, alors `b` est un diviseur de `a`
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` autrement.

La fonction `diviseur(a, b)`

- `b` est un diviseur de `a` si le reste de la division de `a` par `b` est `0`
- Si `a % b == 0`, alors `b` est un diviseur de `a`
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` autrement.

```
def diviseur(a, b):
    # On suppose b != 0, a et b sont des valeurs numériques
```

La fonction `diviseur(a, b)`

- b est un diviseur de a si le reste de la division de a par b est 0
- Si `a % b == 0`, alors b est un diviseur de a
- La fonction `diviseur(a, b)` retourne `True` si b est un diviseur de a, et `False` autrement.

```
def diviseur(a, b):
    # On suppose b != 0, a et b sont des valeurs numériques
    if a % b == 0:
        est_diviseur = True
    else:
        est_diviseur = False

    return est_diviseur
```

La fonction `diviseur(a, b)`

- `b` est un diviseur de `a` si le reste de la division de `a` par `b` est `0`
- Si `a % b == 0`, alors `b` est un diviseur de `a`
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` autrement.

Version équivalente (revoir la priorité des opérateurs en cas de doute !) :

```
def diviseur(a, b):
    # b != 0, a et b sont des valeurs numériques
    est_diviseur = a % b == 0

    return est_diviseur
```

La fonction `diviseur(a, b)`

- `b` est un diviseur de `a` si le reste de la division de `a` par `b` est `0`
- Si `a % b == 0`, alors `b` est un diviseur de `a`
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` autrement.

Troisième version équivalente, nous n'avons pas besoin de la variable temporaire :

```
def diviseur(a, b):
    # b != 0, a et b sont des valeurs numériques
    return a % b == 0
```

La fonction `diviseur(a, b)`

- `b` est un diviseur de `a` si le reste de la division de `a` par `b` est `0`
- Si `a % b == 0`, alors `b` est un diviseur de `a`
- La fonction `diviseur(a, b)` retourne `True` si `b` est un diviseur de `a`, et `False` autrement.

Troisième version équivalente, nous n'avons pas besoin de la variable temporaire :

```
def diviseur(a, b):
    # b != 0, a et b sont des valeurs numériques
    return a % b == 0
```

Est-ce nécessaire de définir une fonction qui n'exécute qu'une ligne de code ? Quels sont les avantage et inconvénients ?

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen
- Si `n % 2 == 0`, alors `n` est pair

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen
- Si `n % 2 == 0`, alors `n` est pair
- Réutilisons notre fonction `diviseur(a, b)` !

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen
- Si `n % 2 == 0`, alors `n` est pair
- Réutilisons notre fonction `diviseur(a, b)` !

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen
- Si `n % 2 == 0`, alors `n` est pair
- Réutilisons notre fonction `diviseur(a, b)` !

```
def pair(n):
    # n : valeur numérique
    est_divisible_par_2 = diviseur(n, 2)

    return est_divisible_par_2
```

La fonction `pair(n)`

- Indique si `n` est un nombre pair, retourne un Booléen
- Si `n % 2 == 0`, alors `n` est pair
- Réutilisons notre fonction `diviseur(a, b)` !

Version équivalente :

```
def pair(n):
    # n : valeur numérique
    return diviseur(n, 2)
```

La fonction `impair(n)`

- Indique si `n` est un nombre impair, retourne un Booléen

La fonction `impair(n)`

- Indique si `n` est un nombre impair, retourne un Booléen
- Réutilisons notre fonction `pair(n)` !

La fonction `impair(n)`

- Indique si `n` est un nombre impair, retourne un Booléen
- Réutilisons notre fonction `pair(n)` !

La fonction `impair(n)`

- Indique si `n` est un nombre impair, retourne un Booléen
- Réutilisons notre fonction `pair(n)` !

```
def impair(n):
    # n : valeur numérique
    return not pair(n)
```

Le passage de valeurs par paramètres

- `diviseur(12, 4)`

Le passage de valeurs par paramètres

- `diviseur(12, 4)`
- `diviseur(3 * 4, 2**2)`

Le passage de valeurs par paramètres

- `diviseur(12, 4)`
- `diviseur(3 * 4, 2**2)`
- `diviseur(abs(-12), 3 + 1)`

Le passage de valeurs par paramètres

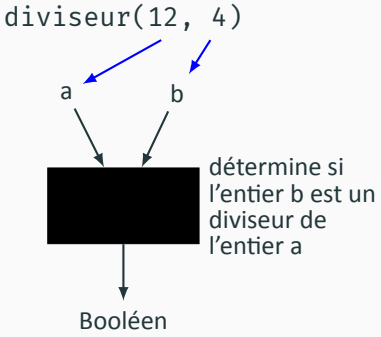
- `diviseur(12, 4)`
- `diviseur(3 * 4, 2**2)`
- `diviseur(abs(-12), 3 + 1)`
- Appel : `nom_de_fonction(expr1, expr2 ,...)`

Le passage de valeurs par paramètres

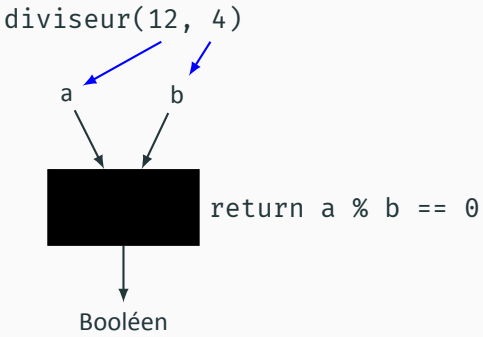
- `diviseur(12, 4)`
- `diviseur(3 * 4, 2**2)`
- `diviseur(abs(-12), 3 + 1)`
- Appel : `nom_de_fonction(expr1, expr2 ,...)`
- Les expressions sont d'abord évaluées, puis les valeurs sont passés en *arguments* à la fonction.

Le passage de valeurs par paramètres

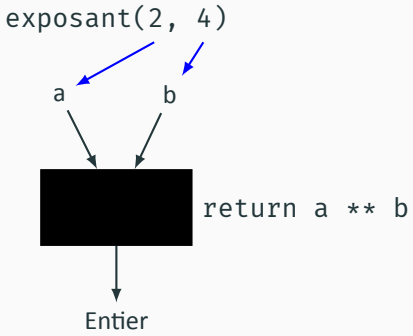
Lors de l'appel de `diviseur(12, 4)`, les valeurs 12 et 4 sont les *arguments* avec lesquels on appelle la fonction. Les *paramètres* a et b prendront les valeurs des arguments pendant l'exécution du code de la fonction.



Compromis : lisibilité, cohérence, efficacité



Fonction `exposant(a, b)` ?




```
def exposant(a, b):
    return a ** b
```

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python
- La fonction sera réutilisée (`premier()`)

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python
- La fonction sera réutilisée (`premier()`)
- La fonction améliorera la lisibilité du code (`diviseur()`)

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python
- La fonction sera réutilisée (`premier()`)
- La fonction améliorera la lisibilité du code (`diviseur()`)
- La fonction permet de continuer la conception de la solution en reportant à plus tard la conception de parties du programme

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python
- La fonction sera réutilisée (`premier()`)
- La fonction améliorera la lisibilité du code (`diviseur()`)
- La fonction permet de continuer la conception de la solution en reportant à plus tard la conception de parties du programme
- La fonction est simple (tient sur un écran normalement)

La décomposition fonctionnelle, lorsque...

- La fonction n'a pas d'équivalent existant (opérateur ou fonction existante)... voir les nombreuses librairies disponibles en Python
- La fonction sera réutilisée (`premier()`)
- La fonction améliorera la lisibilité du code (`diviseur()`)
- La fonction permet de continuer la conception de la solution en reportant à plus tard la conception de parties du programme
- La fonction est simple (tient sur un écran normalement)
- L'expérience vous dira jusqu'à quel niveau il faut décomposer une solution en fonctions

Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs

Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs
- Fonction `carre_et_cube(x)`, qui retourne `x**2` et `x**3`

Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs
- Fonction `carre_et_cube(x)`, qui retourne `x**2` et `x**3`
- Façon d'utiliser la fonction : `a, b = carre_et_cube(4)`

Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs
- Fonction `carre_et_cube(x)`, qui retourne `x**2` et `x**3`
- Façon d'utiliser la fonction : `a, b = carre_et_cube(4)`
- Utiliser ensuite les variables `a` et `b` dans des expressions

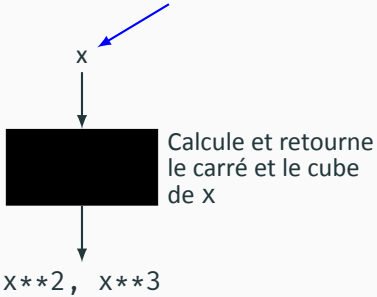
Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs
- Fonction `carre_et_cube(x)`, qui retourne `x**2` et `x**3`
- Façon d'utiliser la fonction : `a, b = carre_et_cube(4)`
- Utiliser ensuite les variables `a` et `b` dans des expressions

Le retour de plus d'une valeur

- Il peut parfois être pratique de définir une fonction qui retourne plusieurs valeurs
- Fonction `carre_et_cube(x)`, qui retourne `x**2` et `x**3`
- Façon d'utiliser la fonction : `a, b = carre_et_cube(4)`
- Utiliser ensuite les variables `a` et `b` dans des expressions

`carre_et_cube(expr)`



Le retour de plus d'une valeur

Déclaration :

```
def carre_et_cube(x):
    return x**2, x**3  # On retourne un couple de valeurs
```

Le retour de plus d'une valeur

Déclaration :

```
def carre_et_cube(x):
    return x**2, x**3  # On retourne un couple de valeurs
```

Utilisation :

```
mon_carre, mon_cube = carre_et_cube(4)

print(mon_carre)
print(mon_cube)
```

Attention au passage d'arguments

Le passage d'une valeur en argument à une fonction **copie** la donnée reçu dans la variable **locale** (le paramètre) de la fonction.

Attention au passage d'arguments

Le passage d'une valeur en argument à une fonction **copie** la donnée reçu dans la variable **locale** (le paramètre) de la fonction.

Par exemple, lors de l'appel `carre_et_cube(3 * 2 - 1)`, la valeur 5 est copiée dans la variable `x` de la fonction.

Attention au passage d'arguments

Certaines données sont trop volumineuses pour procéder ainsi, par exemple : les *listes*.

Attention au passage d'arguments

Certaines données sont trop volumineuses pour procéder ainsi, par exemple : les *listes*.

Alors, elle ne seront pas copiées, et la fonction travaillera sur l'espace mémoire original.

Attention au passage d'arguments

Certaines données sont trop volumineuses pour procéder ainsi, par exemple : les *listes*.

Alors, elle ne seront pas copiées, et la fonction travaillera sur l'espace mémoire original.

Si on modifie la liste dans la fonction, la liste originale sera également modifiée.

Attention au passage d'arguments

Certaines données sont trop volumineuses pour procéder ainsi, par exemple : les *listes*.

Alors, elle ne seront pas copiées, et la fonction travaillera sur l'espace mémoire original.

Si on modifie la liste dans la fonction, la liste originale sera également modifiée.

On y reviendra !

Attention aux définitions *récursives*

```
def pair(n):
    return not impair(n)

def impair(n):
    return not pair(n)
```

Attention aux définitions *récursives*

```
def pair(n):
    return not impair(n)

def impair(n):
    return not pair(n)
```

Que se passe-t-il si on appelle l'une de ces fonctions ?

Attention aux définitions *récursives*

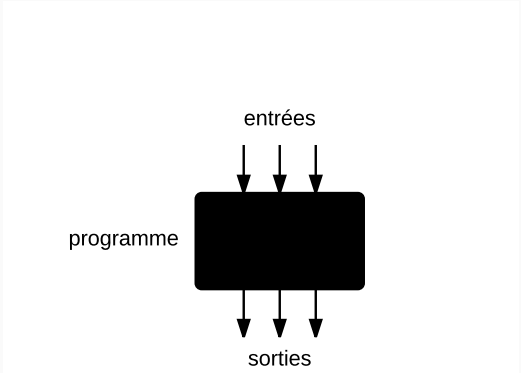
```
def pair(n):
    return not impair(n)

def impair(n):
    return not pair(n)
```

Que se passe-t-il si on appelle l'une de ces fonctions ?

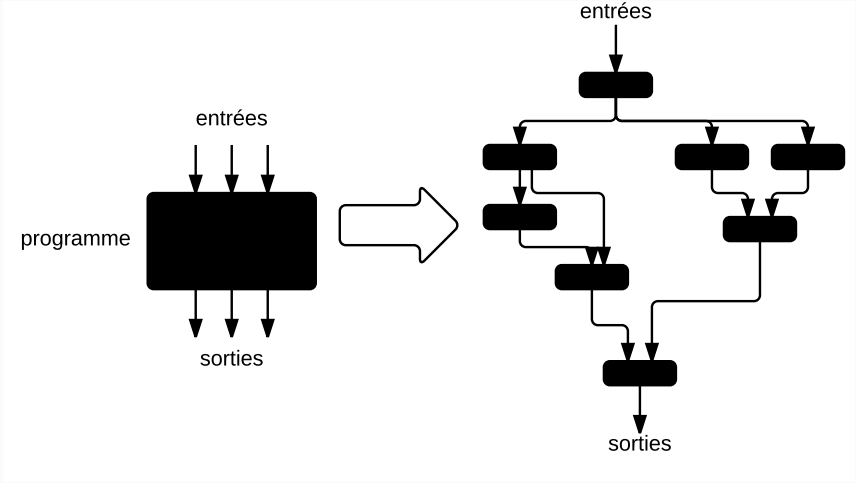
On y reviendra !

- On échafaude les niveaux supérieurs de la solution



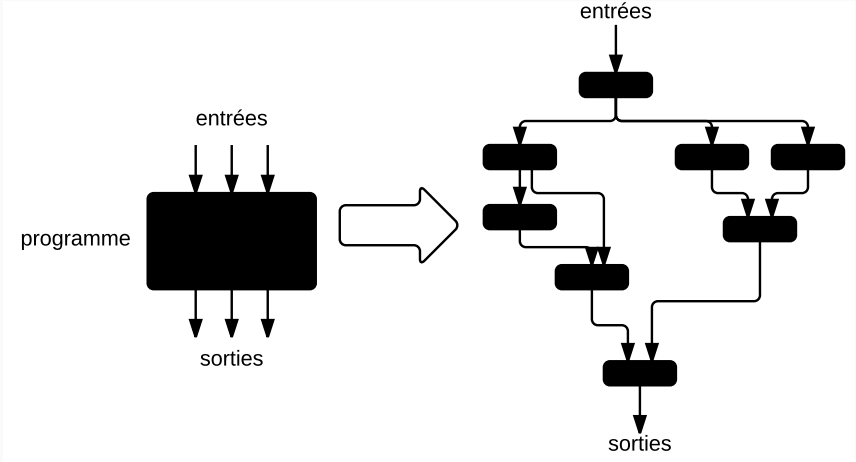
Le développement en équipe de travail

- On échafaude les niveaux supérieurs de la solution



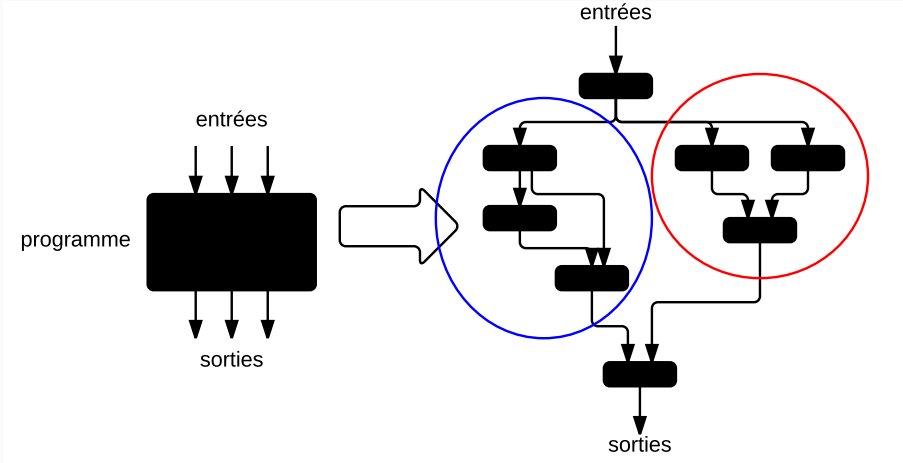
Le développement en équipe de travail

- On échafaude les niveaux supérieurs de la solution
- On découpe la solution en groupes de tâches



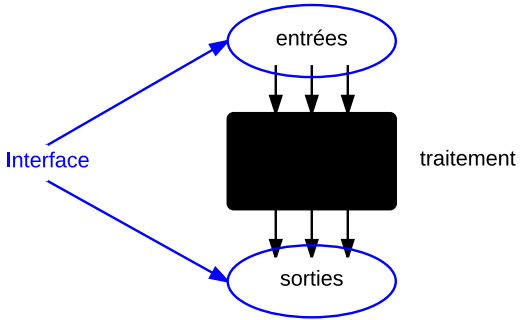
Le développement en équipe de travail

- On échafaude les niveaux supérieurs de la solution
- On découpe la solution en groupes de tâches



Le développement en équipe de travail

- On échafaude les niveaux supérieurs de la solution
- On découpe la solution en groupes de tâches
- On définit l'interface de chaque fonction



Le développement en équipe de travail

- On échafaude les niveaux supérieurs de la solution
- On découpe la solution en groupes de tâches
- On définit l'interface de chaque fonction

```
def premier(n):
    # Détermine si n est un nombre premier
    # n est un entier >= 1
    # retourne un Booléen
```

Pré et post-conditions

Interface d'une fonction =

- Ensemble de conditions devant être vraies avant l'exécution de la fonction, appelées les **pré-conditions**

Pré et post-conditions

Interface d'une fonction =

- Ensemble de conditions devant être vraies avant l'exécution de la fonction, appelées les **pré-conditions**
- Ensemble de conditions devant être vraies après l'exécution de la fonction, appelées les **post-conditions**

Pré et post-conditions

Interface d'une fonction =

- Ensemble de conditions devant être vraies avant l'exécution de la fonction, appelées les **pré-conditions**
- Ensemble de conditions devant être vraies après l'exécution de la fonction, appelées les **post-conditions**

Pré et post-conditions

Interface d'une fonction =

- Ensemble de conditions devant être vraies avant l'exécution de la fonction, appelées les **pré-conditions**
- Ensemble de conditions devant être vraies après l'exécution de la fonction, appelées les **post-conditions**

```
def premier(n):
    # Détermine si n est un nombre premier.
    #
    # Pré-condition : n est un entier >= 1
    # Post-condition : retourne True si n est premier, False autrement
```

Énoncés de travaux pratiques

- À partir d'un problème énoncé, nous allons définir les entrées et les sorties des **programmes** que vous devrez concevoir et implémenter

Énoncés de travaux pratiques

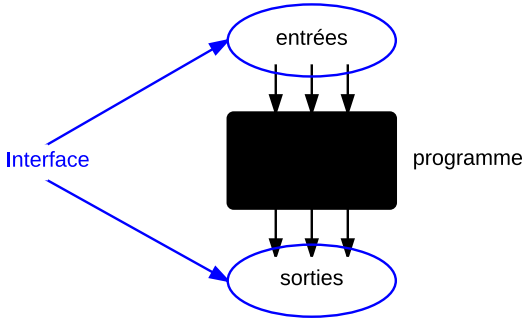
- À partir d'un problème énoncé, nous allons définir les entrées et les sorties des **programmes** que vous devrez concevoir et implémenter
- On définira l'interface du programme attendu

Énoncés de travaux pratiques

- À partir d'un problème énoncé, nous allons définir les entrées et les sorties des **programmes** que vous devrez concevoir et implémenter
- On définira l'interface du programme attendu

Énoncés de travaux pratiques

- À partir d'un problème énoncé, nous allons définir les entrées et les sorties des **programmes** que vous devrez concevoir et implémenter
- On définira l'interface du programme attendu



Vous devrez élaborer une solution

- Vous devrez procéder à une décomposition fonctionnelle

Vous devrez élaborer une solution

- Vous devrez procéder à une décomposition fonctionnelle
- Vous devrez définir les grandes fonctionnalités inhérentes à la résolution du problème

Vous devrez élaborer une solution

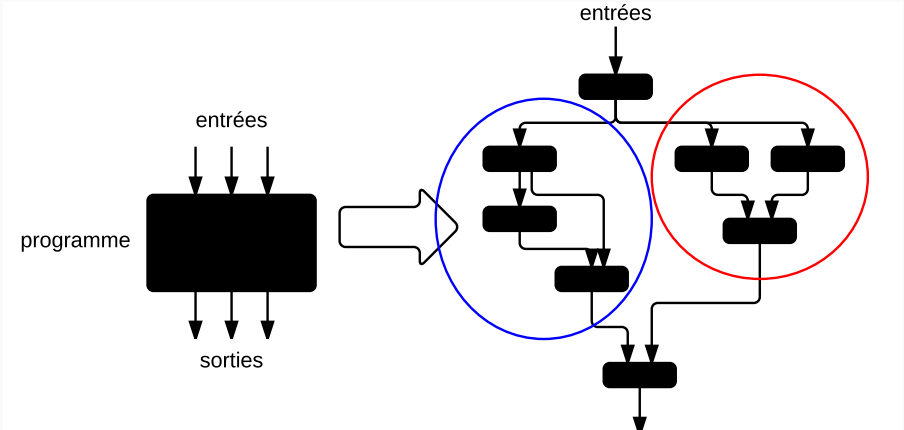
- Vous devrez procéder à une décomposition fonctionnelle
- Vous devrez définir les grandes fonctionnalités inhérentes à la résolution du problème
- Vous devrez définir l'interface de ces supra-fonctions

Vous devrez élaborer une solution

- Vous devrez procéder à une décomposition fonctionnelle
- Vous devrez définir les grandes fonctionnalités inhérentes à la résolution du problème
- Vous devrez définir l'interface de ces supra-fonctions

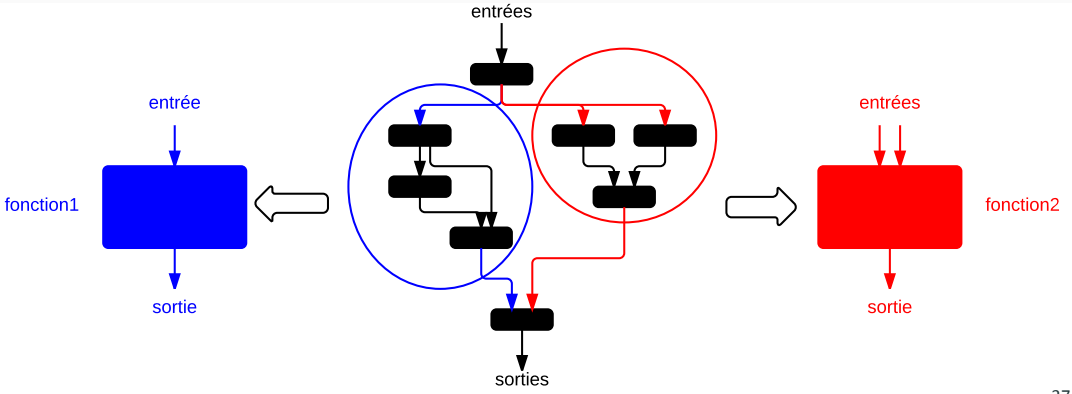
Vous devrez élaborer une solution

- Vous devrez procéder à une décomposition fonctionnelle
- Vous devrez définir les grandes fonctionnalités inhérentes à la résolution du problème
- Vous devrez définir l'interface de ces supra-fonctions



Vous devrez élaborer une solution

- Vous devrez procéder à une décomposition fonctionnelle
- Vous devrez définir les grandes fonctionnalités inhérentes à la résolution du problème
- Vous devrez définir l'interface de ces supra-fonctions



Chacune des équipes devra

- Procéder à une décomposition fonctionnelle de son groupe de tâches

Chacune des équipes devra

- Procéder à une décomposition fonctionnelle de son groupe de tâches
- Définir l'interface des fonctions définies subséquemment

Chacune des équipes devra

- Procéder à une décomposition fonctionnelle de son groupe de tâches
- Définir l'interface des fonctions définies subséquemment
- Respecter l'interface des fonctions utilisées en librairies

Chacune des équipes devra

- Procéder à une décomposition fonctionnelle de son groupe de tâches
- Définir l'interface des fonctions définies subséquemment
- Respecter l'interface des fonctions utilisées en librairies
- Respecter l'interface des fonctions qui vous seront données

Modèle privilégié mais non exclusif

- L'interface d'une fonction devra être bien définie

Modèle privilégié mais non exclusif

- L'interface d'une fonction devra être bien définie
- Elle devra inclure toutes les actions que la fonction réalise

Modèle privilégié mais non exclusif

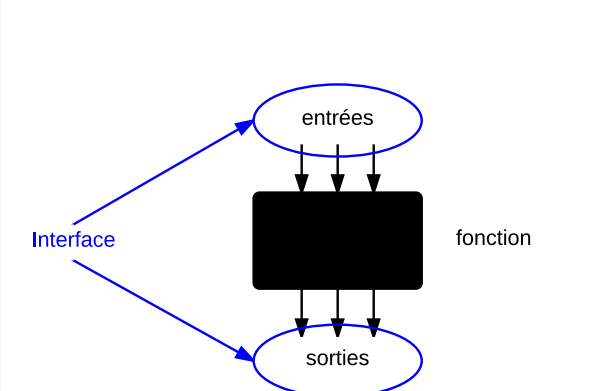
- L'interface d'une fonction devra être bien définie
- Elle devra inclure toutes les actions que la fonction réalise
- On minimisera les actions difficilement identifiables appelées *effets de bord*, car celles-ci changent l'état du système (on y reviendra !)

Modèle privilégié mais non exclusif

- L'interface d'une fonction devra être bien définie
- Elle devra inclure toutes les actions que la fonction réalise
- On minimisera les actions difficilement identifiables appelées *effets de bord*, car celles-ci changent l'état du système (on y reviendra !)

Modèle privilégié mais non exclusif

- L'interface d'une fonction devra être bien définie
- Elle devra inclure toutes les actions que la fonction réalise
- On minimisera les actions difficilement identifiables appelées *effets de bord*, car celles-ci changent l'état du système (on y reviendra !)



Les variables locales et globales

Les variables locales et globales

Objectif

Bien identifier la portée des variables afin de les utiliser correctement

Les variables locales

L'appel d'une fonction crée les variables nécessaires à son exécution, incluant les paramètres de l'en-tête.

```
def premier(n):
    # On suppose que n >= 1

    prem = True  # n est supposé premier
                 # jusqu'à preuve du contraire

    nb = 2  # On "balaie" les entiers entre 2 et n-1
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    # prem contient la réponse
    return prem
```

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
...	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	**True**	Bool
...	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	True	Bool
nb	6114	...	...
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	True	Bool
nb	6114	2	int
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	**False**	Bool
nb	6114	2	int
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	False	Bool
nb	6114	3	int
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	False	Bool
nb	6114	3	int
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Après le retour de la fonction, l'espace mémoire est **libéré**.

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	4	int
prem	6113	False	Bool
nb	6114	3	int
...	6115	...	...

L'exécution de `premier(4)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Après le retour de la fonction, l'espace mémoire est **libéré**.

Nom	Adresse	Contenu	Type
...	6111	...	...
...	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
...	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
...	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	...	...
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	**True**	Bool
...	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	...	...
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	2	int
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	3	int
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	4	int
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	4	int
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Après le retour de la fonction, l'espace mémoire est **libéré**.

Nom	Adresse	Contenu	Type
...	6111	...	...
n	6112	5	int
prem	6113	True	Bool
nb	6114	4	int
...	6115	...	...

L'exécution de `premier(5)`

```
def premier(n):
    prem = True
    nb = 2
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Après le retour de la fonction, l'espace mémoire est **libéré**.

Nom	Adresse	Contenu	Type
...	6111	...	...
...	6112	...	...
...	6113	...	...
...	6114	...	...
...	6115	...	...

Les variables globales

Toutes les **variables définies dans un fichier**, en dehors d'une fonction ou d'une *classe*, sont des variable globales au *module*.

```
nb = 2  # variable globale

def premier(n):
    prem = True
    nb = 2  # variable locale, avec une adresse
            # mémoire différente
    while nb <= n - 1 and prem:
        prem = not diviseur(n, nb)
        nb = nb + 1

    return prem
```

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?
- Cohérence : problème d'accès concurrentiel

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?
- Cohérence : problème d'accès concurrentiel
- Conflit de noms : est-ce qu'on accès à la bonne variable ?

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?
- Cohérence : problème d'accès concurrentiel
- Conflit de noms : est-ce qu'on accès à la bonne variable ?
- Effet de bord possiblement non souhaitable

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?
- Cohérence : problème d'accès concurrentiel
- Conflit de noms : est-ce qu'on accès à la bonne variable ?
- Effet de bord possiblement non souhaitable

Les variables globales

Problèmes potentiels de :

- Longévité : quelles variables sont gardées en mémoire pendant l'exécution du programme ?
- Cohérence : problème d'accès concurrentiel
- Conflit de noms : est-ce qu'on accès à la bonne variable ?
- Effet de bord possiblement non souhaitable

À éviter sauf dans des cas particuliers !

Documenter une fonction

Documenter une fonction

Plusieurs standards existent pour documenter correctement une fonction. Des outils existent pour par la suite générer automatiquement la documentation d'un programme. Nous vous proposons d'utiliser les *Google Style Python Docstrings* :

`http://sphinxcontrib-napoleon.readthedocs.org/en/latest/example_google.html#example-google`

Documenter une fonction

```
def ma_fonction(x, y, z=0):
    """Cette fonction fait telle et telle action.

    Args:
        x (float): description courte.
            Une description longue optionnelle peut être
            ajoutée, à un niveau d'indentation plus bas.
        y (str): description courte de y.
        z (int, optional): description de z (défaut: 0).

    Returns:
        bool: True si telle condition est respectée,
        False autrement.

        Le type est optionnel, et la description peut
        être faite sur plusieurs lignes, avec le même
        niveau d'indentation.
    """
```

Les modules

Objectif

Pouvoir développer des programmes Python par groupes de tâches connexes, groupées dans un même fichier appelé *module*.

Les modules

- Un *module* est un fichier qui *encapsule* des fonctions et des variables ayant rapport entre elles et offrant des services complémentaires, dont le tout peut être vu comme cohérent

Les modules

- Un *module* est un fichier qui *encapsule* des fonctions et des variables ayant rapport entre elles et offrant des services complémentaires, dont le tout peut être vu comme cohérent
- On peut créer des modules afin de bien structurer un programme, i.e. en groupant des fonctions de même utilité ou d'utilité complémentaire

Les modules

- Un *module* est un fichier qui *encapsule* des fonctions et des variables ayant rapport entre elles et offrant des services complémentaires, dont le tout peut être vu comme cohérent
- On peut créer des modules afin de bien structurer un programme, i.e. en groupant des fonctions de même utilité ou d'utilité complémentaire
- En Python, un module est un fichier, et le nom du module est le nom du fichier, par exemple `mon_module.py`

Les modules

- Un *module* est un fichier qui *encapsule* des fonctions et des variables ayant rapport entre elles et offrant des services complémentaires, dont le tout peut être vu comme cohérent
- On peut créer des modules afin de bien structurer un programme, i.e. en groupant des fonctions de même utilité ou d'utilité complémentaire
- En Python, un module est un fichier, et le nom du module est le nom du fichier, par exemple `mon_module.py`
- On se sert d'un module de la même façon que vu en travaux dirigés

```
from mon_module import nom_fonction
from autre_module import *  # pratique, mais à éviter
```

Les modules

- Un *module* est un fichier qui *encapsule* des fonctions et des variables ayant rapport entre elles et offrant des services complémentaires, dont le tout peut être vu comme cohérent
- On peut créer des modules afin de bien structurer un programme, i.e. en groupant des fonctions de même utilité ou d'utilité complémentaire
- En Python, un module est un fichier, et le nom du module est le nom du fichier, par exemple `mon_module.py`
- On se sert d'un module de la même façon que vu en travaux dirigés

```
from mon_module import nom_fonction
from autre_module import *  # pratique, mais à éviter
```

- Tout module doit débuter par un commentaire explicatif.

Les modules : Exemple

dessins_tortue.py :

```
"""Module regroupant des fonctions de
dessins diverses pour une tortue.
"""
from turtle import *

def carre(taille, couleur):
    """Fonction qui dessine un carré de taille
    et de couleur déterminées.
    """
    color(couleur)
    c = 0
    while c < 4:
        forward(taille)
        right(90)
        c = c + 1
```

Les modules : Exemple

principal.py :

```
"""Fichier principal (point d'entrée du programme)."""

from dessins_tortue import carre

up()  # relever le crayon
goto(-150, 50)  # reculer en haut à gauche
i = 0
while i < 10:
    down()  # abaisser le crayon
    carre(25, 'red')  # tracer un carré
    up()  # relever le crayon
    forward(30)  # avancer plus loin
    i = i + 1

input("Tapez sur une touche pour quitter...")
```

En résumé

- La décomposition fonctionnelle et la structuration d'un programme en modules favorisent :

En résumé

- La décomposition fonctionnelle et la structuration d'un programme en modules favorisent :
  - L'organisation d'un programme pour en favoriser la compréhension par un lecteur externe

En résumé

- La décomposition fonctionnelle et la structuration d'un programme en modules favorisent :
  - L'organisation d'un programme pour en favoriser la compréhension par un lecteur externe
  - La réutilisation logicielle

En résumé

- La décomposition fonctionnelle et la structuration d'un programme en modules favorisent :
  - L'organisation d'un programme pour en favoriser la compréhension par un lecteur externe
  - La réutilisation logicielle
  - Le développement modulaire

En résumé

- La décomposition fonctionnelle et la structuration d'un programme en modules favorisent :
  - L'organisation d'un programme pour en favoriser la compréhension par un lecteur externe
  - La réutilisation logicielle
  - Le développement modulaire
- La documentation des modules et fonctions sera une tâche importante pour un informaticien

Les tests unitaires

Les tests unitaires

Objectif

Pouvoir s'assurer que chaque traitement et module développé fonctionne convenablement selon les normes et considérations établies lors de la conception du programme.

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`
  - `premier(9)` devrait retourner `False`

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`
  - `premier(9)` devrait retourner `False`
- Il faut tester plusieurs cas :

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`
  - `premier(9)` devrait retourner `False`
- Il faut tester plusieurs cas :
  - Les plus variés possibles

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`
  - `premier(9)` devrait retourner `False`
- Il faut tester plusieurs cas :
  - Les plus variés possibles
  - Les cas *extrêmes* et les cas normaux

Les tests unitaires

- Les tests unitaires assurent le fonctionnement de chaque *unité* de votre programme
  - Les fonctions
  - La cohérence entre les fonctions (modules)
- On vérifie que certains appels de fonctions retournent bien le résultat attendu :
  - `premier(3)` devrait retourner `True`
  - `premier(9)` devrait retourner `False`
- Il faut tester plusieurs cas :
  - Les plus variés possibles
  - Les cas *extrêmes* et les cas normaux
- Une fois les tests codés on peut ainsi les réutiliser automatiquement

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests
- Plus tard, si on trouve un bogue dans la fonction :

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests
- Plus tard, si on trouve un bogue dans la fonction :
  - On améliore le test (pour vérifier le bogue non vu initialement)

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests
- Plus tard, si on trouve un bogue dans la fonction :
  - On améliore le test (pour vérifier le bogue non vu initialement)
  - On corrige le bogue

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests
- Plus tard, si on trouve un bogue dans la fonction :
  - On améliore le test (pour vérifier le bogue non vu initialement)
  - On corrige le bogue
  - On ré-exécute le test **complet**

Utilisation des tests unitaires

- On code d'abord les tests qui définissent le comportement de la fonction voulue, ou mieux, les tests et la fonction sont codés par deux personnes différentes !
- On code ensuite la fonction qui doit valider tous les tests
- Plus tard, si on trouve un bogue dans la fonction :
  - On améliore le test (pour vérifier le bogue non vu initialement)
  - On corrige le bogue
  - On ré-exécute le test **complet**
- On ne doit pas faire échouer un test qui fonctionnait avant la correction d'un bogue !

Choix des tests unitaires

- Les plus variés possibles

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :
    - Avec des valeurs *au milieu* de la plage de valeurs

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :
    - Avec des valeurs *au milieu* de la plage de valeurs
  - Cas limites :

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :
    - Avec des valeurs *au milieu* de la plage de valeurs
  - Cas limites :
    - Avec les extrêmes (minimum et maximum)

Choix des tests unitaires

- Les plus variés possibles
- Couvrir le plus de cas différents
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :
    - Avec des valeurs *au milieu* de la plage de valeurs
  - Cas limites :
    - Avec les extrêmes (minimum et maximum)
- Plus il y a de tests, plus le code sera robuste !

Code des tests unitaires

```python
def square(x):
    """ Met x au carré."""
    return x * x

if __name__ == '__main__':
    assert square(4) == 16
    assert square(2) == 4
    assert square(0) == 0
    assert square(-2) == 4
    assert square(1) == 1
    assert square(1000) == 1000
    # Attention : assert square(0.1) == 0.01 va échouer!
    print(square(0.1))  # 0.010000000000000002
    assert abs(square(0.1) - 0.01) < 0.00000001
```

`if __name__ == '__main__'` ???!

- Le corps principal d'un script Python constitue une entité dont le nom réservé est `'__main__'`

`if __name__ == '__main__'` ???!

- Le corps principal d'un script Python constitue une entité dont le nom réservé est `'__main__'`
- L'exécution d'un module commence toujours avec la première instruction de cette entité

`if __name__ == '__main__'` ???!

- Le corps principal d'un script Python constitue une entité dont le nom réservé est `'__main__'`
- L'exécution d'un module commence toujours avec la première instruction de cette entité
- Tout repose sur la variable `__name__` qui existe dès le lancement de l'interpréteur. Si le module est importé, alors cette variable prend comme valeur le nom du module.

`if __name__ == '__main__'` ???!

- Le corps principal d'un script Python constitue une entité dont le nom réservé est `'__main__'`
- L'exécution d'un module commence toujours avec la première instruction de cette entité
- Tout repose sur la variable `__name__` qui existe dès le lancement de l'interpréteur. Si le module est importé, alors cette variable prend comme valeur le nom du module.
- Le code dans le bloc `'__main__'` ne sera donc exécuté que si l'on exécute le module en tant que programme. Si le module est importé par un autre module, ce code ne sera jamais exécuté.

À retenir

- Il faut **documenter** vos programmes

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction
- Un module est un fichier avec l'extension `.py`

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction
- Un module est un fichier avec l'extension `.py`
  - Permet d'organiser les différents constituants d'un programme en modules indépendants

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction
- Un module est un fichier avec l'extension `.py`
  - Permet d'organiser les différents constituants d'un programme en modules indépendants
  - Favorise la réutilisation du code

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction
- Un module est un fichier avec l'extension `.py`
  - Permet d'organiser les différents constituants d'un programme en modules indépendants
  - Favorise la réutilisation du code
- On peut **importer** les éléments d'un module à partir d'un autre module

À retenir

- Il faut **documenter** vos programmes
  - chaque fichier
  - chaque fonction
- Un module est un fichier avec l'extension `.py`
  - Permet d'organiser les différents constituants d'un programme en modules indépendants
  - Favorise la réutilisation du code
- On peut **importer** les éléments d'un module à partir d'un autre module
- Il faut **tester** les modules adéquatement

Lectures, travaux et exercices

Plan des activités du module 3

- Chapitres 5, 6 et 7 de G. Swinnen (on prend de l'avance !)
- Travaux dirigés : Encapsulation en fonctions

Questions ?