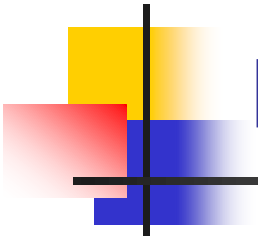


# Programmation

- Notion de programme
- Quelques instructions du langage Python

# Qu'est-ce qu'un programme?

- C'est une suite d'instructions écrites dans un langage de programmation compréhensible par l'ordinateur.
- Cela permet à l'ordinateur d'appliquer un algorithme.
- Exemple : afficher les diviseurs de $n$

```
fonction affiche_diviseurs(n)
  si n > 0 alors
    pour tout entier i entre 1 et n faire
      si n est divisible par i alors
        afficher i
      finsi
    finpour
  finsi
```

```
def affiche_diviseurs(n):
  if n > 0:
    for i in range(1, n+1):
      if n % i == 0:
        print i
```

Remarque : `n % i` donne le reste de la division de *n* par *i*

# L'affectation :
## ranger une valeur dans une variable

- L'ordinateur effectue les instructions dans l'ordre. L'ordre des instructions est donc très important.
- Une variable désigne un emplacement dans lequel on peut mémoriser une valeur. Une variable a un nom.
- En python, le symbole = n'a pas la même signification qu'en mathématique. Il signifie calculer la valeur à droite du symbole =, et la ranger dans la variable dont le nom se trouve à gauche. C'est l'affectation.
- Exemples :

```
i = 1
x = 2*i+1
x = x+i
```

- Exercice 1.1.[1 … 6]

# Expression booléennes

- Une expression booléenne est une expression qui n'a que 2 valeurs possibles :

  `True` (Vrai)

  `False` (False)

- Les tests sont des expressions booléennes

  égalité `x==y`

  inégalité `x!=y`

  comparaison `x<=y` `x>y` etc.

- On peut les combiner avec des opérateurs : `and`, `or` et `not`

# Division entière

- Le quotient entier `q` de deux entiers `a` et `b` positifs et le reste `r` sont définis par :

  `a = bq+r` avec `0<=r<b`

- En python :

  Quotient entier de a par b : **`a//b`**

  Reste ou modulo : **`a%b`**

Exemple : `19//5` donne `3`

`19%5` donne `4`

Exercice 1.1.8

# Notion de fonction

En maths : soit la fonction $f : x \rightarrow 2x^2 + 1$

décrit la façon de calculer **f(x)** à partir de la donnée **x**

En Python :

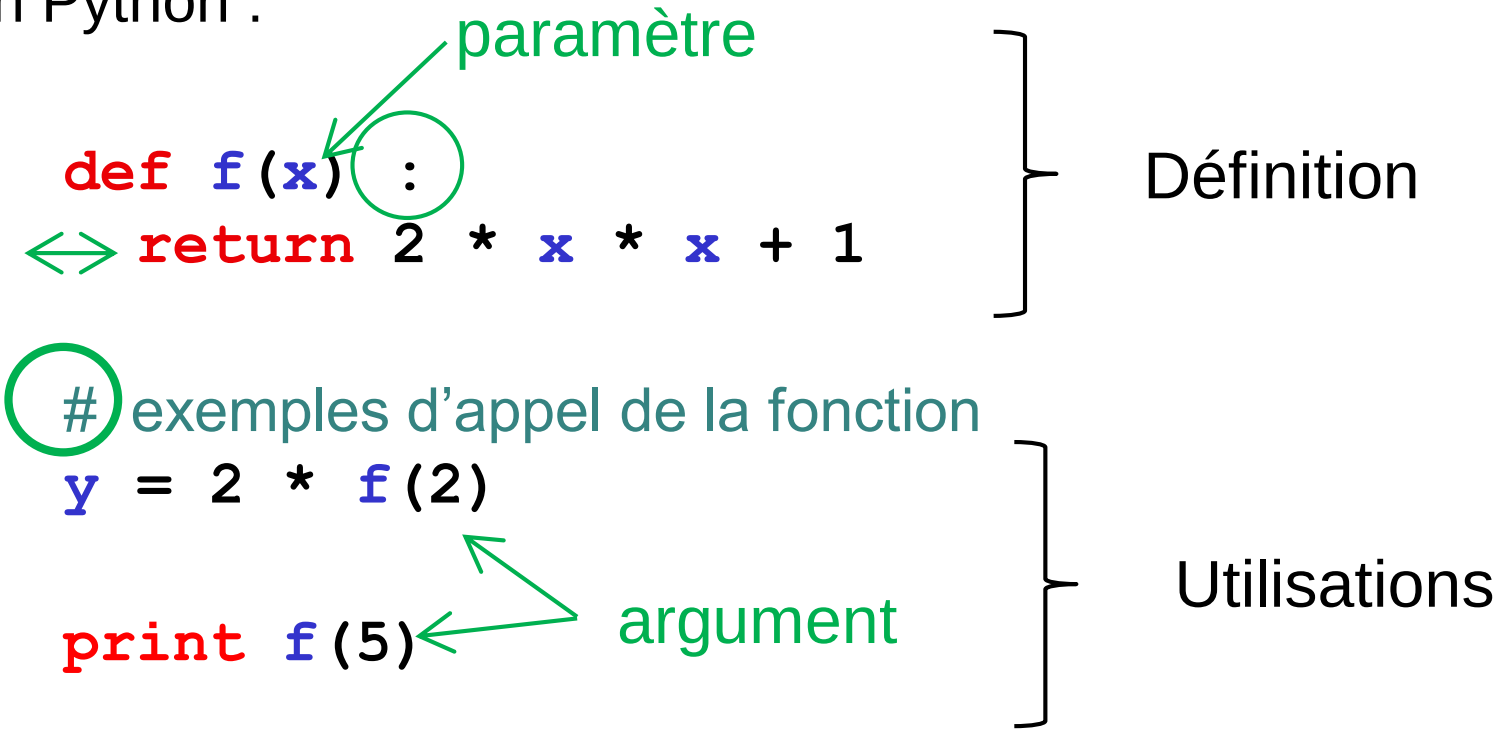
```
def f(x) :
  return 2 * x * x + 1

# exemples d'appel de la fonction
y = 2 * f(2)

print f(5)
```

# Notion de fonction

En maths : soit la fonction $f : x \rightarrow 2x^2 + 1$

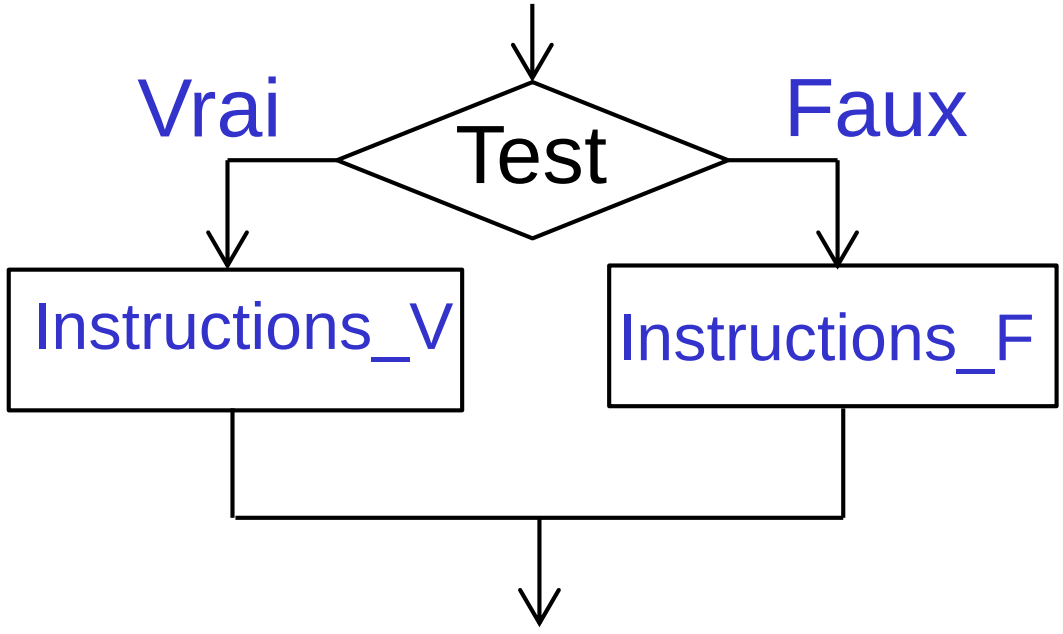
décrit la façon de calculer `f(x)` à partir de la donnée `x`

En Python :



# Instruction conditionnelle **if…: else :…**

Instruction conditionnelle si… alors… sinon…



# Instruction conditionnelle **if…: else :…**

Instruction conditionnelle si… alors… sinon…

```
if Test :
   Instructions_V
else :
   Instructions_F
```

# Instruction conditionnelle **if…: else :…**

Instruction conditionnelle si… alors… sinon…

Exemple

```
i=10

x=6

if i > x :
    print "test VRAI"
    print i, "est supérieur à", x
else :
    print "test FAUX"
    print i, "n'est pas supérieur à", x
```

Attention à l'indentation !

# Instruction conditionnelle **if…: else :…**

Instruction conditionnelle si… alors… sinon…

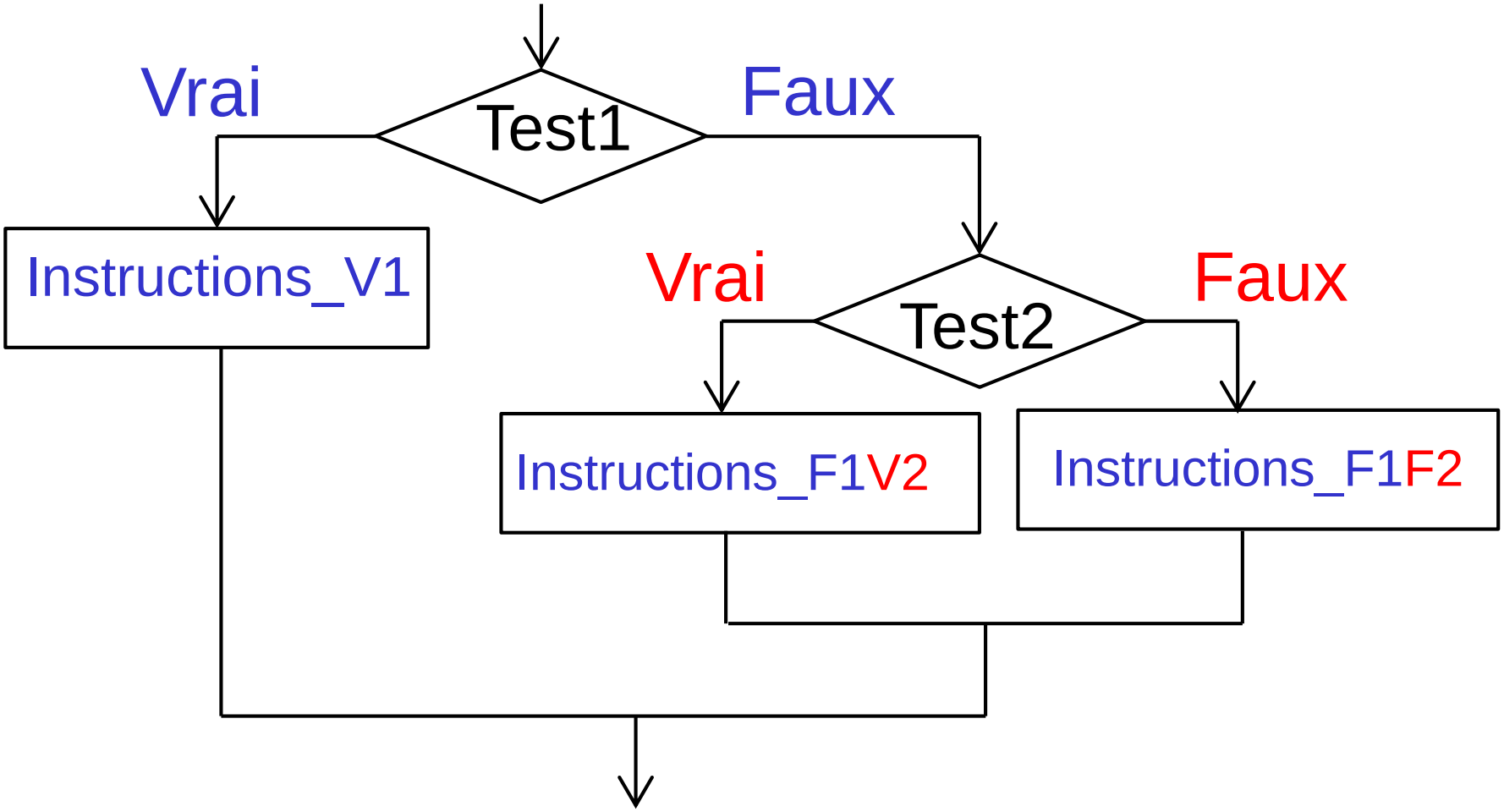
Exemple

```
i=10

x=6

if i > x :
↔print "test VRAI"
↔print i, "est supérieur à", x
else :
↔print "test FAUX"
↔print i, "n'est pas supérieur à", x
```

Attention à l'indentation !

# Instruction conditionnelle**if…: elif…: else :…**

On peut « imbriquer » les **`if`**… **`:else`**…



# Instruction conditionnelle**if…: elif…: else :…**

On peut « imbriquer » les **if… :else…**

```
if x<0 :
  print "negatif "
else :
  if x%2 ==0:
    print "pair "
  else :
    print "impair "
```

```
if x<0 :
  print "negatif "
elif x%2 ==0:
  print "pair "
else :
  print "impair "
```

# Instruction conditionnelle**if…: elif…: else :…**

On peut « imbriquer » les **if… :else…**

```
if x<0 :
↔print "negatif "
else :
↔if x%2 ==0:
↔↔print "pair "
↔else :
↔↔print "impair "
```

```
if x<0 :
↔print "negatif "
elif x%2 ==0:
↔print "pair "
else :
↔print "impair "
```

# Listes d'éléments

On peut manipuler des listes d'éléments :

Exemples :

```
[3,7,4,1]
[``d'',``f'',``e'']
```

```
6 in [3,7,4,1] => False
4 in [3,7,4,1] => True
```

La fonction **len** retourne la taille de la liste.

```
Len([3,7,4,1]) => 4
```

# Listes d'éléments

La fonction **range** permet de construire des listes d'éléments consécutifs

# liste des entiers ∈ [a,b[ par pas de c

```
range(a,b,c):
```

Remarque: `range(a,b)` ⇔ `range(a,b,1)`

`range(b)` ⇔ `range(0,b,1)`

Exemple :

```
list( range(1,9,1)) =>[1,2,3,4,5,6,7,8]
list( range(1, 9, 2)) => [1,3,5,7]
```

# Répétition : **for … in … :**

Répétition : pour… parcourant la liste ….

La boucle **`for`** permet de parcourir les éléments d'une liste

```
# parcourir et afficher les entiers de 0 à 9
for i in range(10) :
    print (i)
```

On peut imbriquer les boucles :

```
for i in range(10):
  for j in range(3):
    print(i,j)
```

# Répétition : **for … in … :**

Répétition : pour… parcourant la liste ….

La boucle **`for`** permet de parcourir les éléments d'une liste

```
# parcourir et afficher les entiers de 0 à 9
for i in range(10) :
    print (i)
```

On peut imbriquer les boucles :

```
for i in range(10):
    for j in range(3):
        print(i,j)
```

# Répétition : **while… :**

Répétition : tant que … faire…

```
i=10
while i > 0 :
  print i
  i = i - 1
```

Ou :

```
while i > 0 :
  i = i - 1
  print i
```

Attention à l'ordre des instructions !

# Répétition : **while… :**

Répétition : tant que … faire…

```
i=10
while i > 0 :
↔print i
↔i = i - 1
```

Ou :

```
while i > 0 :
↔i = i - 1
↔print i
```

Attention à l'ordre des instructions !

# Quelques pièges à éviter !

- Affichage ou sortie (**`print`** ou **`return`**)
- Position du **`return`** (dans la boucle ou après la boucle)

```
def f(x):
  for i in range(x):
    print i

def g(x):
  for i in range(x):
    return i

def h(x):
  for i in range(x):
    print i
  return i
```

Quel sera le résultat de :

**`f`**, **`g`** et **`h`** pour **`x=3`** ?