- Boucle while
- Boucle for
- Le type “ensemble”
- L’instruction break
- Notion de fonction
- Variables locales / globales
- Récursion

Logique et approche mathématique de la programmation, cours 2
M. Rigo

Boucle “while” : tant que *condition* est satisfaite, effectuer le corps de la boucle

In [1]:

```
import random
mystere = random.randint(1, 100) # choix aléatoire d'un nombre entre 1 et 100
entree = 0
print("Devinez le nombre mystère, quelle est votre proposition ?")

while entree != mystere:
    entree = input()
    if entree.isnumeric():
        entree = int(entree)
        if entree < mystere:
            print("Trop petit ! Une autre proposition ?")
        elif entree > mystere:
            print("Trop grand ! Une autre proposition ?")
    else:
        print("Vous n'avez pas entré un nombre, recommencez !")

print("Félicitations ! Le nombre mystère était ",mystere)
```

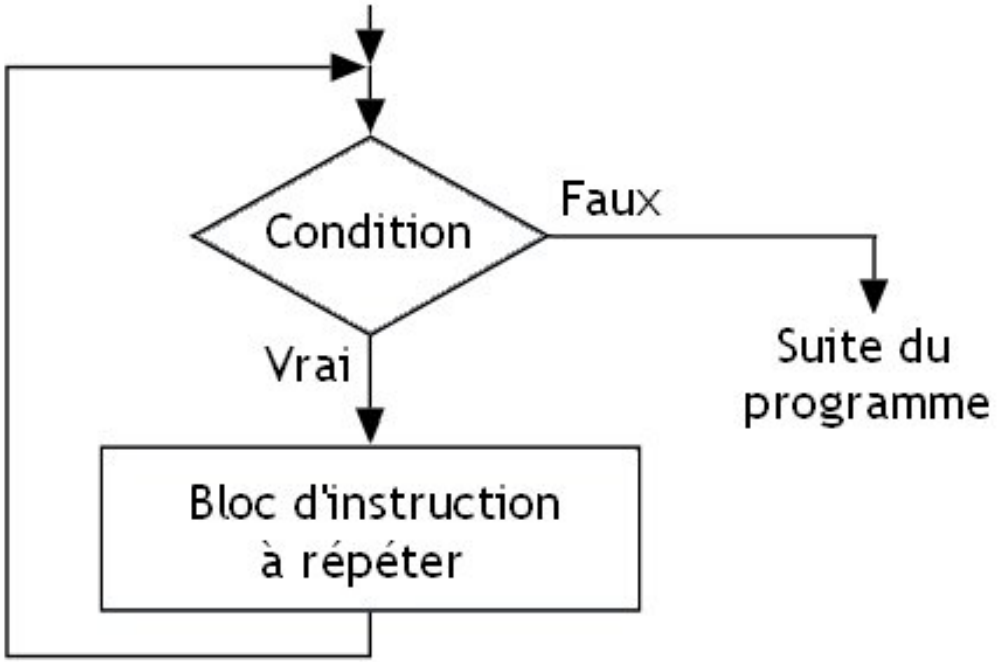
Variante : on gère les exceptions ...

```
import random
mystere = random.randint(1, 100) # choix aléatoire d'un nombre entre 1 et 10
entree = 0
print("Devinez le nombre mystère, quelle est votre proposition ?")

while entree != mystere:
    entree = input()
    try:
        entree = int(entree)
    except:
        print("entrée non valide, recommencez")
    else:
        if entree < mystere:
            print("Trop petit ! Une autre proposition ?")
        elif entree > mystere:
            print("Trop grand ! Une autre proposition ?")

print("Félicitations ! Le nombre mystère était ",mystere)
```

Variante : on compte le nombre d’essais...

```
import random
mystere = random.randint(1, 100) # choix aléatoire d'un nombre entre 1 et 100
entree = 0
print("Devinez le nombre mystère, quelle est votre proposition ?")
essai = 0

while entree != mystere:
    entree = input()
    try:
        entree = int(entree)
    except:
        print("entrée non valide, recommencez")
    else:
        essai +=1
        if entree < mystere:
            print("Trop petit ! Une autre proposition ?")
        elif entree > mystere:
            print("Trop grand ! Une autre proposition ?")

print("Félicitations ! Le nombre mystère était ",mystere)
print("Vous avez trouvé en ",essai," essais")
```

Une boucle “tant que” (while), on répète … tant que la condition est satisfaite




```
In [1]: while 1==0:
            print("aaa")
```

On n’entre pas dans la boucle ; le programme n’affiche rien.

```
In [2]: x=5
        while x!=0:
            print(x)
            x -= 1
```

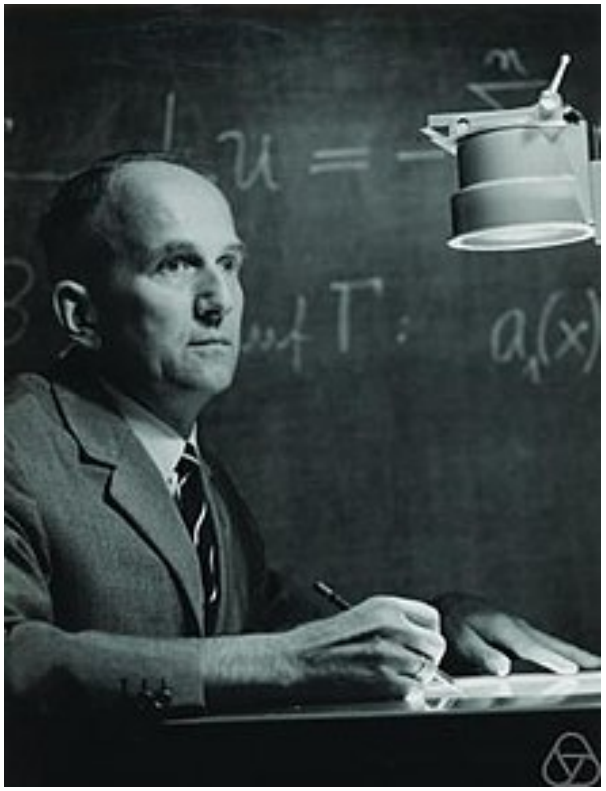
```
5
4
3
2
1
```

Le problème de Syracuse, la conjecture de Collatz, ou encore le *3x+1 problem*, ...

ABSTRACT. The $3x+1$ problem concerns iteration of the map $T : \mathbb{Z} \to \mathbb{Z}$ given by

$$T(x) = \begin{cases} \dfrac{3x+1}{2} & \text{if } x \equiv 1 \pmod 2 \ . \\ \\ \dfrac{x}{2} & \text{if } x \equiv 0 \pmod 2 \ . \end{cases}$$

The $3x+1$ Conjecture asserts that each $m \geq 1$ has some iterate $T^{(k)}(m) = 1$. This is an annotated bibliography of work done on the $3x+1$ problem and related problems from 1963 through 1999. At present the $3x+1$ Conjecture remains unsolved.



Le problème de Syracuse, la conjecture de Collatz, ou encore 3x+1, ...

On va enregistrer l'orbite dans une liste

```
x = 125
l=[]

while x != 1:
    if x %2 == 0:
        x = x//2
    else:
        x = 3*x+1
    l.append(x)

print(l)
print("nombre d'itérations ",len(l))
```

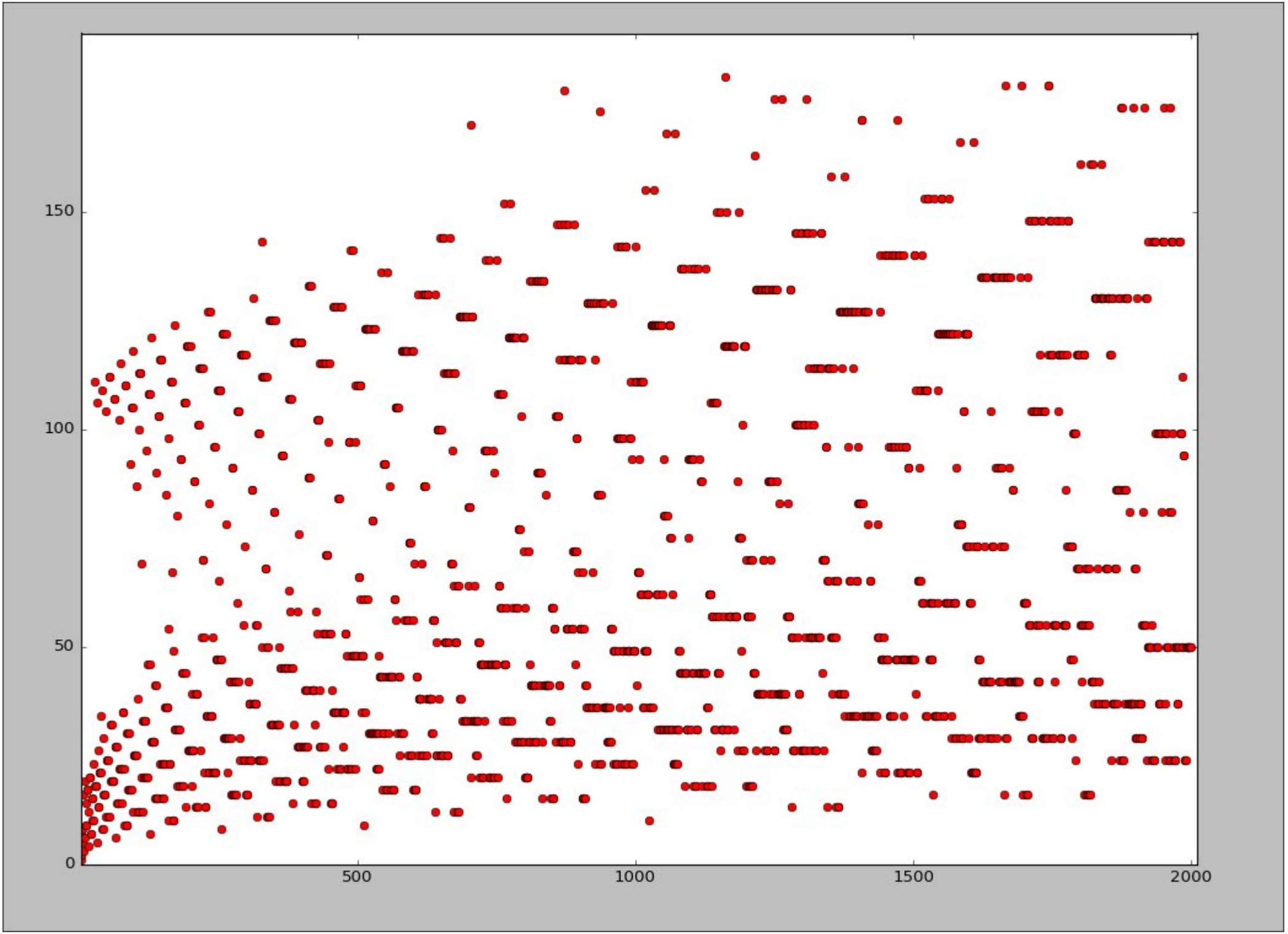
```
[376, 188, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 2
33, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566,
283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1
367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184,
92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
nombre d'itérations  108
```

Apparition d’une boucle for...

```
tot=[]

for i in range(2,200):
    x=i
    l=[]
    while x!=1:
        if x %2 == 0:
            x=x//2
        else:
            x=3*x+1
        l.append(x)

    tot.append(len(l)) # on mémorise le nombre d'itérations
print(tot)
```

```
[1, 7, 2, 5, 8, 16, 3, 19, 6, 14, 9, 9, 17, 17, 4, 12, 20, 20, 7, 7, 15, 15, 10, 23, 10, 111, 18, 18, 18, 106, 5, 26,
13, 13, 21, 21, 21, 34, 8, 109, 8, 29, 16, 16, 16, 104, 11, 24, 24, 24, 11, 11, 112, 112, 19, 32, 19, 32, 19, 19, 107,
107, 6, 27, 27, 27, 14, 14, 14, 102, 22, 115, 22, 14, 22, 22, 35, 35, 9, 22, 110, 110, 9, 9, 30, 30, 17, 30, 17, 92, 1
7, 17, 105, 105, 12, 118, 25, 25, 25, 25, 25, 87, 12, 38, 12, 100, 113, 113, 113, 69, 20, 12, 33, 33, 20, 20, 33, 33,
20, 95, 20, 46, 108, 108, 108, 46, 7, 121, 28, 28, 28, 28, 28, 41, 15, 90, 15, 41, 15, 15, 103, 103, 23, 116, 116, 116
, 23, 23, 15, 15, 23, 36, 23, 85, 36, 36, 36, 54, 10, 98, 23, 23, 111, 111, 111, 67, 10, 49, 10, 124, 31, 31, 31, 80,
18, 31, 31, 31, 18, 18, 93, 93, 18, 44, 18, 44, 106, 106, 106, 44, 13, 119, 119, 119, 26, 26, 26, 119]
```


```python
import matplotlib.pyplot as plt

nombre=2000
tot=[]

for i in range(2,nombre):
    x=i
    l=[]
    while x!=1:
        if x %2 == 0:
            x=x//2
        else:
            x=3*x+1
        l.append(x)

    tot.append(len(l)) # on mémorise le nombre d'itérations

plt.plot(range(2,nombre), tot, 'ro')
plt.axis([2, nombre+10, 0, max(tot)+10])
plt.show()
```


150
100
50
0
500
1000
1500
2000

Une boucle “for” permet de passer en revue les éléments d’un objet de type “séquentiel”
Comme une liste, un t-uple, un ensemble, une chaîne, un dictionnaire

```
for x in [1,3,'a',17]:
    print(x)
```

```
1
3
a
17
```

```
for x in range(2,5):
    print(x*x)
```

```
4
9
16
```

```
temp=0
for x in range(1,11):
    temp += x
print(temp)
```

```
55
```

Une boucle “for” permet de passer en revue les éléments d’un objet de type “séquentiel”
Comme une liste, un t-uple, un ensemble, une chaîne, un dictionnaire

```
x=[1,2,3]
y=[-2,7,3]

temp=0
for i in range(0,len(x)):
    temp+= x[i]*y[i]

print(temp)
```

```
c="bonjour"
for a in c:
    print(2*a)
```

```
bb
oo
nn
jj
oo
uu
rr
```

```
c="bonjour"
d=""
for a in c:
    d += 2*a
print(d)
```

```
bboonnjjoouurr
```

Le type “ensemble”

```
s1 = {1,2,"a",6,6} # c'est la notation mathématique
```

```
s1 # on a bien un ensemble (pas de répétition)
```

```
{1, 2, 6, 'a'}
```

```
s2 = set([1,2,3,3,4]) # on change le type
```

```
s2
```

```
{1, 2, 3, 4}
```

```
s1.union(s2)
```

```
{1, 2, 3, 4, 6, 'a'}
```

```
s1.issubset(s2)
```

```
False
```

```
s1.intersection(s2)
```

```
{1, 2}
```

```
s1.symmetric_difference(s2)
```

```
{3, 4, 6, 'a'}
```

```
s1.add(77)
print(s1)
```

```
{1, 2, 6, 77, 'a'}
```

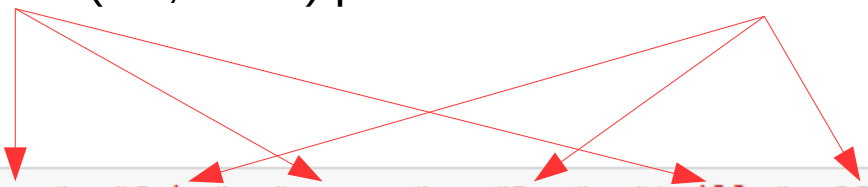
Le type “ensemble”

```
for x in s1:
    print(x)
```

```
1
2
6
77
a
```

Le type “dictionnaire”

Un dictionnaire est une collection d’objets indexés, l’avantage est que l’index (i.e., la clé) pour accéder aux valeurs est choisi par le programmeur



```
dico = { "nom": "John", "prenom" : "Doe", "taille": 183}
for i in dico:
    print(i)
```

```
nom
prenom
taille
```

```
dico["nom"] = "Jane"
```

```
for i in dico:
    print(dico[i])
```

```
Jane
Doe
183
```

Ajout d’une entrée

```
dico["poids"] = 71
```

```
print(dico["poids"])
```

```
71
```

Il existe de nombreuses méthodes : values, items, pop, popitem, del, len, copy, ...

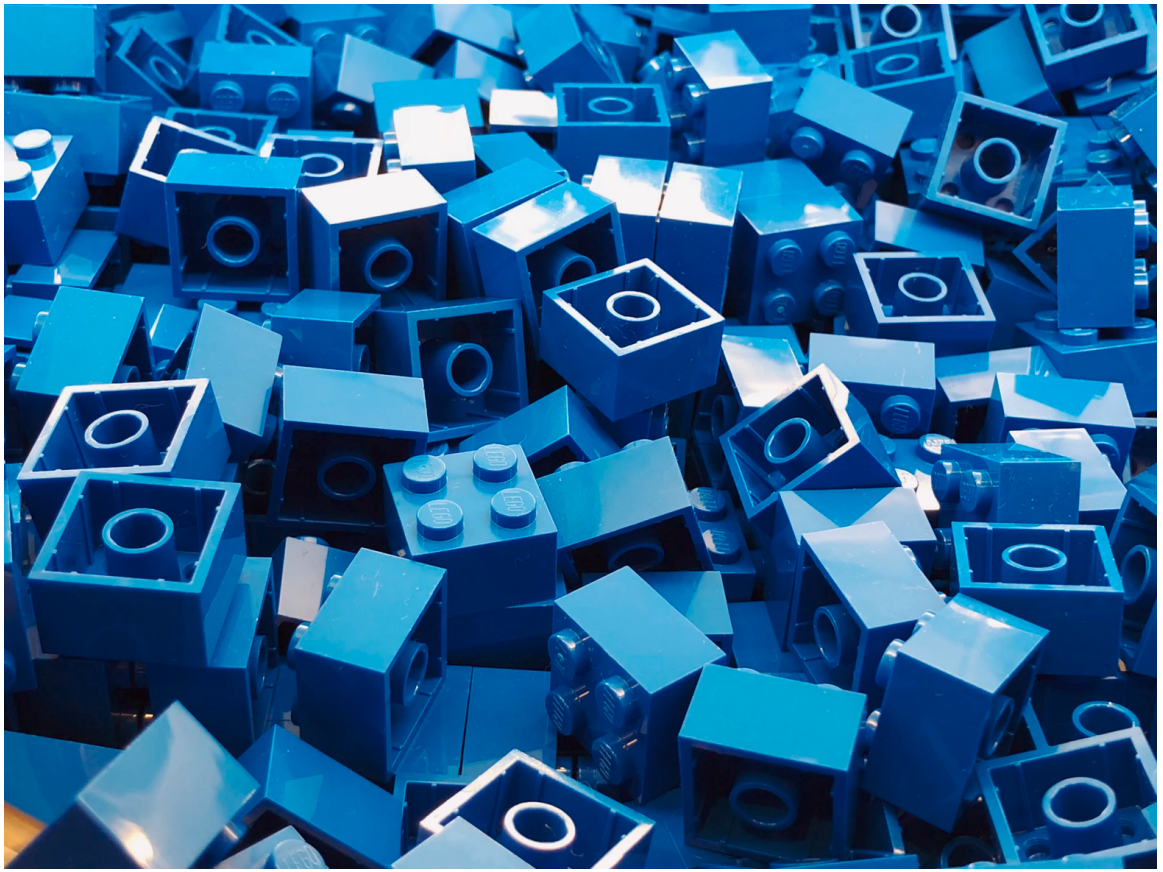
La commande “break” force la sortie d’une boucle (while ou for)

```
while True:
    x = input("entrer quelque chose :")
    if x == "0":
        break
    else:
        print(x,"n'est pas '0'...")
```

```
x=[1,7,12,4,3,4,7,9,11,0]
p=3

for i in range(0,len(x)):
    if x[i]==p:
        break

print(i)

if x[i]==p:
    print("première occurence de",p,"en",i)
```

Définir des fonctions (ou sous-programmes) au sein d'un programme

- Avoir des procédures *réutilisables* (au sein d'un programme, pour plusieurs projets)
- Avoir un code plus lisible
- Découper un problème en problèmes plus simples – *abstraction, modularité*
- Quand une fonction est bien écrite/testée,
  plus besoin d'en connaître le "contenu"
- Peuvent avoir des paramètres, etc.



Algorithme d'Eulide...

$$
\begin{aligned}
b &= a\,q_1 + r_1, & 0 < r_1 < a \\
a &= r_1\,q_2 + r_2, & 0 < r_2 < r_1 \\
r_1 &= r_2\,q_3 + r_3, & 0 < r_3 < r_2 \\
&\vdots & \\
r_{j-2} &= r_{j-1}\,q_j + r_j, & 0 < r_j < r_{j-1} \\
r_{j-1} &= r_j\,q_{j+1}, & r_j \neq 0.
\end{aligned}
$$

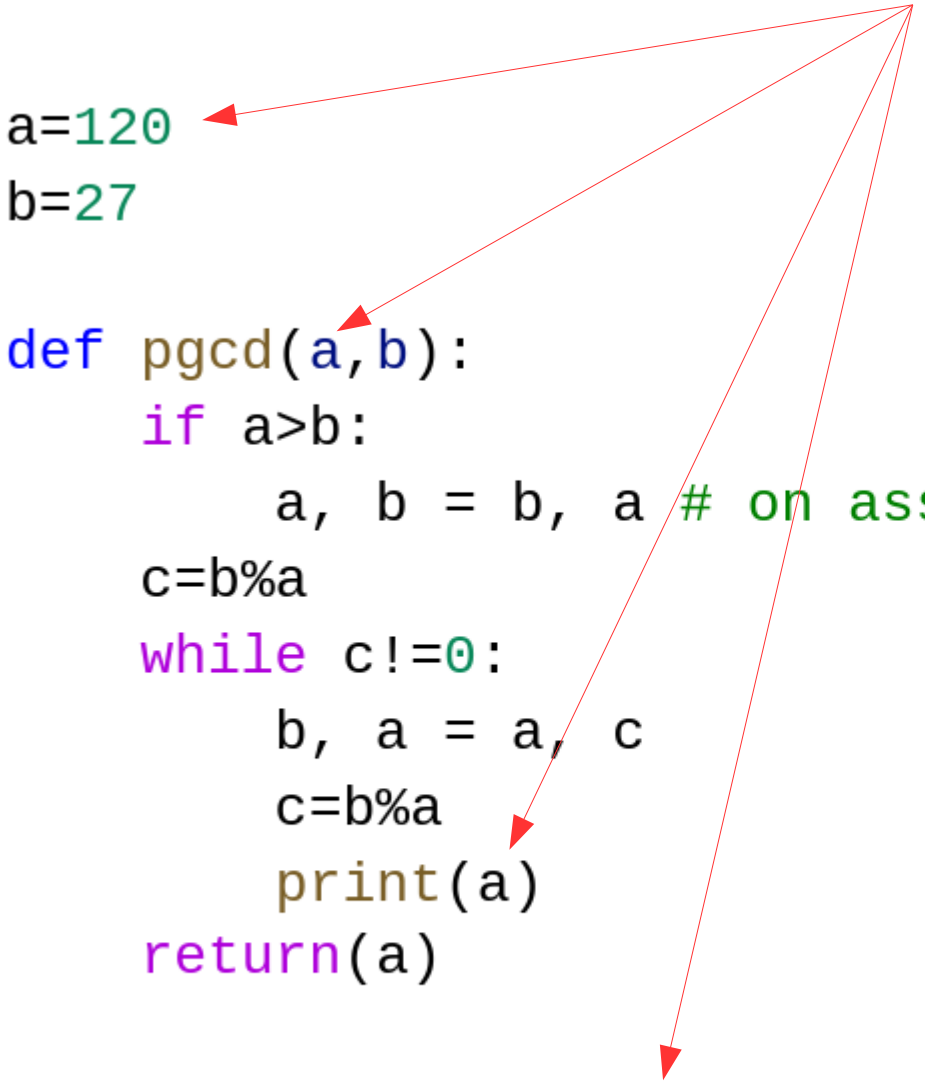
```
b=300
a=121
c=b%a

while c!=0:
    print(b,"=",b//a,".",a,"+",c)
    b, a = a, c # affectations multiples
    c=b%a

print(a)
```

Définir une fonction

```
x=120
y=27

def pgcd(a,b):
    if a>b:
        a, b = b, a # on assure b>=a
    c=b%a
    while c!=0:
        b, a = a, c
        c=b%a
    return(a)

print("le pgcd de",x,"et",y,"=",pgcd(x,y))
```

La fonction “renvoie” quelque chose...

Cela va-t-il poser problème ? Variables locales...



```
a=120
b=27

def pgcd(a,b):
    if a>b:
        a, b = b, a # on assure b>=a
    c=b%a
    while c!=0:
        b, a = a, c
        c=b%a
        print(a)
    return(a)

print("le pgcd de",a,"et",b,"=",pgcd(a,b))
```

Variable globale

Variable locale (n'existe qu'au sein de la fonction)

```
# Premier programme
a = 2

def test():
    a = 3
    print("à l'intérieur de test: ", a)

test()

# Deuxième programme
a = 2

def test():
    print("à l'intérieur de test", a)

test()
print("après test", a)
```

Il n'y a pas de variable locale, donc fait référence à la variable globale

```python
# Troisième programme
a = 2

def test():
    a = 3
    print("à l'intérieur de test", a)

test()
print("Après test", a)
```

A quoi fait-on référence ?

```python
# Quatrième programme
a = 2

def test():
    global a
    a = 3
    print("à l'intérieur de test", a)

test()
print("après test", a)
```

On impose l'utilisation
de la variable globale...

```python
import math

x=[1,2,3]
y=[-2,7,3]

def produit_scalaire(x,y):
    temp=0
    for i in range(0,len(x)):
        temp+= x[i]*y[i]
    return(temp)

def norme(x):
    return(math.sqrt(produit_scalaire(x,x)))

print(produit_scalaire(x,y), norme(x))
```

```python
def syracuse(x):
    if x %2 == 0:
        x=x//2
    else:
        x=3*x+1
    return(x)

def nbre_iterations(x):
    i=0
    while x!=1:
        x=syracuse(x)
        i+=1
    return(i)

x=100
while x!=1:
    x=syracuse(x)
    print(x)

print(nbre_iterations(100))
```

Une définition récursive

miroir(*abcde*) = *e* + miroir(*bcde*)

Cas de base : le miroir de “” = “”

F(n+2) = F(n+1) + F(n)
Avec des conditions initiales *F(0)=1* et *F(1)=2*

N! = N.(N-1)!
et *0!=1*

```python
def miroir(mot):
    if len(mot)==0:
        return(mot)
    else:
        return(mot[-1]+miroir(mot[0:-1]))

une_chaine="bonjour"
print(une_chaine," : ",miroir(une_chaine))

def test_palindrome(mot):
    if mot==miroir(mot):
        return(True)
    else:
        return(False)

print(test_palindrome("kayak"))
une_chaine="kayak"
if test_palindrome(une_chaine):
    print(une_chaine,"est un palindrome")
else:
    print(une_chaine,"n'est pas un palidrome")
```

```
def fibonacci_rec(n):
    """ rend le n-ieme terme de la suite de Fibonacci """
    if n==0:
        return 1
    elif n==1:
        return 2
    else:
        return fibonacci_rec(n-1)+fibonacci_rec(n-2)
```

Cela fonctionne mais ATTENTION :
comparer le nombre d'appels à la fonction
avec le nombre de valeurs à calculer …

Il faut parfois trouver le bon compromis.

```
import time
nbre_appels=0

def fibonacci_rec(n):
    global nbre_appels
    nbre_appels += 1
    if n==0:
        return 1
    elif n==1:
        return 2
    else:
        return fibonacci_rec(n-1)+fibonacci_rec(n-2)

start = time.time()
print("-- version récursive --")
print(fibonacci_rec(30))
end = time.time()
print("temps de calcul ", end - start)
print("nombre d'appels ",nbre_appels)
```

→ 2178309, temps de calcul 0.62 sec., nombre d'appels 2692537

On stocke les valeurs calculées pour ne pas les recalculer à chaque nouvel appel !

```
fib=[1,2]
nbre_appels2=0

def fibonacci_2(n):
    global nbre_appels2
    nbre_appels2 += 1
    # si la liste est trop courte, calculer un élément
    if len(fib)<=n:
        fib.append(fib[-1]+fib[-2])
        fibonacci_2(n)
    return fib[n]

start = time.time()
print("-- version dynamique --")
print(fibonacci_2(30))
end = time.time()
print("temps de calcul ", end - start)
print("nombre d'appels ",nbre_appels2)
```

→ 2178309, temps de calcul 3.55e-05, nombre d'appels 30

Encore une variante (pour ne pas “multiplier” les appels à la fonction)

```
fib=[1,2]

def fibonacci_3(n):

# on construit d'une traite les éléments manquants
    if len(fib)<=n:
        for i in range(0,n-len(fib)+1):
            fib.append(fib[-1]+fib[-2])
    return fib[n]
```

Exercice : “améliorer” la fonction de syracuse

Ecrire une fonction qui à *n* associe le nombre d’itérations nécessaires pour arriver à 1

Stocker à chaque fois la valeur obtenue

Ainsi, si syracuse(n) nécessite le calcule de syracuse(k) et que celle-ci est déjà connue, reprendre la valeur stockée

Comparer les performances des deux approches (sans et avec stockage)

Challenge : implémenter l'écriture en base b...

Quels nombres sont des palindromes écrits en base 2 et 3 ?

```
? IntegerDigits
```

IntegerDigits[n] gives a list of the decimal digits in the integer n.
IntegerDigits[n, b] gives a list of the base b digits in the integer n.
IntegerDigits[n, b, len] pads the list on the left with zeros to give a list of length len.
IntegerDigits[n, MixedRadix[$blist$]] uses the mixed radix with list of bases $blist$. ≫

```
IntegerDigits[2375, 10]
```

```
{2, 3, 7, 5}
```

```
IntegerDigits[2375, 2]
```

```
{1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1}
```

```python
def integer_digits(n,b):
    """ renvoie la liste des chiffres de n
    en base 2"""
    liste=[]
    while n>0:
        liste = liste + [n%b]
        n=n//b
    return(liste)

l=integer_digits(19,2)
print(l)
miroir=l[::-1]
print(miroir)

for i in range(1,100):
    l=integer_digits(i,2)
    miroir=l[::-1]
    if l==miroir:
        print(i,":",l)
```

```
def test_palindrome(n,b):
    l=integer_digits(n,b)
    miroir=l[::-1]
    if l==miroir:
        return True
    else:
        return False

for i in range(1,10000000):
    if test_palindrome(i,2) and test_palindrome(i,3):
        print(i)
```

1, 6643, 1422773, 5415589 et après ?
http://oeis.org/A060792 (uniquement 9 < 3^66)

Il serait peut-être plus efficace de générer les palindromes dans une base et faire uniquement le test dans l'autre base...

Bérczes, Attila(H-LAJO-IM); Ziegler, Volker(A-OAW-RIC)
On simultaneous palindromes. (English summary)
J. Comb. Number Theory 6 (2014), no. 1, 37–49.
11A63 (11Y55)

Sums of Palindromes: an Approach via Automata

Aayush Rajasekaran, Jeffrey Shallit, and Tim Smith

Recently, Cilleruelo, Luca, & Baxter proved, for all bases $b \geq 5$, that every natural number is the sum of at most 3 natural numbers whose base-b representation is a palindrome. However, the cases $b = 2, 3, 4$ were left unresolved.

We prove, using a decision procedure based on automata, that every natural number is the sum of at most 4 natural numbers whose base-2 representation is a palindrome. Here the constant 4 is optimal. We obtain similar results for bases 3 and 4, thus completely resolving the problem.

```
? FromDigits
```

FromDigits[*list*] constructs an integer from the list of its decimal digits.

FromDigits[*list*, *b*] takes the digits to be given in base *b*.

FromDigits[*list*, MixedRadix[*blist*]] uses the mixed radix with list of bases *blist*.

FromDigits["*string*"] constructs an integer from a string of digits.

FromDigits["*string*", "Roman"] constructs an integer from Roman numerals. ≫