Itération et algorithmes

Ce chapitre revisite les itérations et introduit des algorithmes.

Réaffectation

Une première affectation de variable crée cette variable et y associe cette valeur.

In [85]:
```
x = 5
print(x)
```

```
5
7
```

Une réaffectation y associe une nouvelle valeur.

In [4]:
```
x = 7
print(x)
```

```
7
```

La variable y est affecté avec la valeur de x. Quand x change, y garde sa vaelur.

In [8]:
```
y = x
```

In [7]:
```
x = 3
print(y)
```

```
7 3
```

Mettre à jour les variables

Une opération fréquente est l'incrémentation d'une variable.

In [11]:
```
print(x)
x = x + 1
print(x)
```

```
5
6
```

La décrémentation d'une variable, c'est quand une valeur est soustrait du valeur de la variable.

In [12]:
```
print(x)
x = x - 1
print(x)
```

```
6
5
```

Il existe un raccourci pour écrire cette fonction

In [13]:
```
print(x)
x += 2
print(x)
```

```
5
7
```

En fait, toutes les opérateurs (`+-*/%`) possèdent ce raccourcis

In [21]:
```
x = 10
print(x)
x += 1
print(x)
x //= 2
print(x)
x **= 3
print(x)
x %=4
print(x)
```

```
10
11
5
125
1
```

L'instruction while

Dans un programme doit souvent répéter une séquence d'instructions. On appelle cette répétition une itération. La connotation avec itération est qu'il y a une petite modification à chaque répétition.

Le mot-clé `while` introduit une boucle. Tandis que la condition est vraie, le corps du `while` est répété.

In [24]:
```
def count_down(n):
    while n > 0:
        print(n)
        n -= 1
    print('Finish')

count_down(3)
```

```
3
2
1
Finish
```

Pour cerains calcul c'est difficile à dire si une boucle se termine.

In [40]:
```
def sequence(n):
    while n != 1:
        print(n, end=', ')
        if n % 2 == 0:
            n = n // 2
        else:
            n = n * 3 + 1
    return n
```

La condition de la boucle est `n != 1` , donc la boucle continue jusqu'à ce que `n` soit 1.

In [41]:
```
for n in range(2, 20):
    print(sequence(n))
```

```
2, 1
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2
, 1
10, 5, 16, 8, 4, 2, 1
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
12, 6, 3, 10, 5, 16, 8, 4, 2, 1
13, 40, 20, 10, 5, 16, 8, 4, 2, 1
14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
16, 8, 4, 2, 1
17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8,
4, 2, 1
19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8
, 4, 2, 1
```

break - interrompre une boucle

Parfois on doit pouvoir interrompre une boucle depuis l'interieur.

In [1]:
```
while True:
    line = input('>')
    if line == 'end':
        break
    else:
        print(line)

print('END')
```

```
>this
this
>is
is
>end
END
```

Racines carrées

Une façon de calculer les racines carrées est d'utiliser un algorithme itérative, par exemple la méthode de Newton. Supposons que nous voulions calculer la racine carrée de a. A partir d'un estimation x, nous obtenons une meilleure estimation en utilisant la formule

$$y = \frac{x + a/x}{2}$$

In [5]:
```
a = 4
x = 3
y = (x + a/x) / 2
```

Nous pouvons définir une fonction qu'il suffit d'appliquer quelques fois

In [13]:
```
def newton(x, a):
    return (x + a/x) / 2
```

In [8]:
```
x = newton(x, a); x
```

Out[8]: 2.1666666666666665

In [9]:
```
x = newton(x, a); x
```

Out[9]: 2.0064102564102564

In [10]:
```
x = newton(x, a); x
```

Out[10]: 2.0000102400262145

Lorsque `x==y` nous pouvons arrêter. Voici une boucle qui commence avec la première estimation et améliore jus'qu'à ce quelle ne change plus.

In [15]:
```
a = 4
x = 3
while True:
    print(x)
    y = newton(x, a)
    if y==x:
        break
    x = y
```

```
3
2.1666666666666665
2.0064102564102564
2.0000102400262145
2.0000000000262146
2.0
```

In [19]:
```
def root(a):
    x = a
    while True:
        print(x)
        y = newton(x, a)
        if y==x:
            return x
        x = y
```

In [25]:
```
root(3)
```

```
3
2.0
1.75
1.7321428571428572
1.7320508100147274
1.7320508075688772
```

Out[25]:
```
1.7320508075688772
```

Exercices

ex 1 - esilon

Encapsulez l'algorithme de Newton pour la racine carrée dans une fonction `mysqrt` qui prend x comme paramètre et renvoie une estimation de la racine carré.

Ecrivez une fonction `test_racine` qui compare cette valeur avec `math.sqrt(x)`

In [29]:
```
def mysqrt(x):
    a = x
    while True:
        y = (x + a/x)/2
        if y == x:
            return x
        x = y

mysqrt(2)
```

Out[29]:
```
1.414213562373095
```

In [54]:
```
import math

cols = '{:<3}{:<24}{:<24}{:<24}'
print(cols.format('x', 'mysqrt(x)', 'math.sqrt(x)', 'diff'))
for x in range(1, 10):
    a = mysqrt(x)
    b = math.sqrt(x)
    print(cols.format(x, a, b, b-a))
```

```
x  mysqrt(x)              math.sqrt(x)            diff
1  1                      1.0                     0.0
2  1.414213562373095      1.4142135623730951      2.220446049250313
e-16
3  1.7320508075688772     1.7320508075688772      0.0
4  2.0                    2.0                     0.0
5  2.23606797749979       2.23606797749979        0.0
6  2.449489742783178      2.449489742783178       0.0
7  2.6457513110645907     2.6457513110645907      0.0
8  2.82842712474619       2.8284271247461903      4.440892098500626
e-16
9  3.0                    3.0                     0.0
```

In [57]:
```
2**-52
```

Out[57]:
```
2.220446049250313e-16
```

La valeur est la plus petit incrément d'une valeur flottante à 64 bits avec 1 signe, 53 bits de mantisse et 10bits d'exposant

ex 2 - eval

La fonction interne **eval** prend une chaîne de caractères et l'évalue comme du code Python. Ecrivez une fonction qui invite l'utilisateur à faire une saisie, évalue cette expression et affiche le résultat jusqu'à ce que l'utilisateur entre le mot *fini*.

In [58]:
```
while True:
    exp = input('>')
    if exp == 'fini':
        break
    print(eval(exp))
```

```
>1+23
24
>math.sqrt(23)
4.795831523312719
>fini
```

ex 3 - série infinie pour approximer pi

Le mathématicien Srinivasa Ranujan a trouvé une série infinie qui peut être utilisée pour générer une approximation numérique de $1/\pi$.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1193 + 26390k)}{(k!)^4 396^{4k}}$$

In [70]:
```
def estimate_pi(k):
    res =  * math.factorial(4*k) *
    return res

estimate_pi(0)
```

Out[70]:
```
0.31830987844047015
```

In [84]:
```
from math import sqrt, factorial, pi

sum = 0.0
k = 0
a = 2 * sqrt(2) / 9801

while True:
    b = factorial(4*k) * (1103 + 26390 * k) / (factorial(k)**4 * 396**
    sum += b
    if b < 1e-15:
        break
    k += 1

res = 1 / (a * sum)
print(res, pi)
```

```
3.141592653589793 3.141592653589793
```