Python au lycée pour les profs

- Saison 1 -

Le programme Algo-Prog de la Saison 1 (2nde+)

- Objectifs de cet enseignement Algo-Prog :
  - Acquisition d'une première **démarche algorithmique**.
  - La notion de **fonction** : données —> résultat
  - La **programmation** comme **rédaction** d'un **texte** précis destiné à être **exécuté par une machine** (ordinateur, tablette, smartphone…).

Le logiciel Python

- Vous pouvez installer le logiciel **Python 3** sur votre :

ordinateur

www.python.org
ou Anaconda...

tablette/smartphone

Pydroid 3 (Android)
Pythonista 3 (iPad)
Carnets (iPad)

Pourquoi apprendre à **programmer** ?

- Pour **faire faire par une machine** des calculs trop pénibles à la main.

  Nombres premiers : 2 3 5 7 11 13 17 19 23…

  Combien de nombres premiers jusqu'à 1000 ?

  codes secrets…

- Pour **faire faire par un ordinateur** des calculs répétitifs…

  Gérer des statistiques de données journalières.

- Pour (soi-même) apprendre à raisonner en termes d'algorithmes.

  Comment savoir si 5645393 est un nombre premier ?

- Pour (soi-même) apprendre à penser en termes d'**efficacité** : estimer le **temps de calcul** et la **mémoire** consommée par la machine.

- Pour revenir aux sources des mathématiques : le **CALCUL**.

computer ← ***computation*** ←

(ordinateur) *(en anglais)*

- Parce qu'il y a l'Intelligence Artificielle, les robots, etc. et que la France ne veut pas passer à côté de l'Histoire, surtout en Sciences…

Les objets utilisés en Python

• Le programmeur va travailler sur des objets divers : des nombres, des textes, etc. Chaque ***objet*** appartient à une ***classe*** (ou ***type***) qui définit les *opérations* légales sur ses objets.

• Sur des **nombres** comme `-35` ou `3.14` on pourra faire des additions, des divisions, de la trigonométrie, etc.

• Sur des **textes** comme `'Vendredi 13 mars'`, on pourra faire des recherches, des remplacements, des transformations, etc.

• Sur des **collections** d'objets : stocker, rechercher, compter, etc.

• Avec une **tortue** : tracer des courbes, produire de l'art numérique, etc.

```
>>> n = 15770634024286 * 463289751
>>> n                          #  un nombre
7306373110223588892786
>>> s = str(n)          # nombre ⟶ texte
```

```
>>> s                  #  un texte
'7306373110223588892786'
>>> len(s)             # sa longueur
22                     # 22 chiffres
```

```
def somme(n) :    # pour n ≥ 0 entier, renvoie 1+2+...+n
    res = 0
    for k in range(1,n+1) :
        res = res + k
    return res

print('1+2+…+100 =', somme(100))
```

nombres

Juste le goût de la chose...

```
1+2+…+100 = 5050
```

```
def chifs(s) :    # extraction des chiffres de la chaîne s
    res = ''
    for i in range(len(s)) :
        if ord('0') <= ord(s[i]) <= ord('9') :
            res = res + s[i]
    return res

print(chifs('H2SO4 (5 fois)'))
```

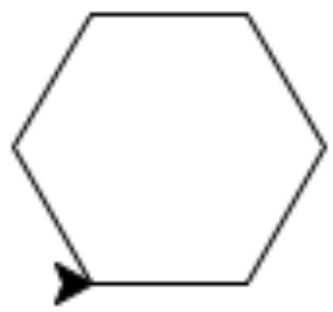
textes

```
'245'
```

```
from turtle import *

for k in range(6) :    # hexagone régulier
    forward(30)
    left(60)
```

graphisme tortue



Des variables pour nommer des objets Python

- Notre langage de programmation manipule des **variables**.

```
x     y     a     x1     point2     parent_de
```

- Lorsque le nom d'une variable est un mot contenant plusieurs lettres, ne le débutez **pas** par une **M**ajuscule, svp (c'est réservé aux *classes*)...

```
x     X     a     A1     L     Point2     Parent_de
```

- L'opérateur d'**affectation** = permet de donner une valeur à une variable, ou de modifier sa valeur. Cette valeur sera un objet Python.

```
>>> a = 2
>>> x1 = 3 + a
>>> x1
5
```

a ← 2
x1 ← 3 + a
langage naturel

```
>>> a = a + 1
>>> a
3
>>> x1
5
```

a ← a + 1
devient égal à

☜ *n'a pas changé !*

~~x - 3 = 2~~

aucun sens !

Ne confondez pas = et ==

La **valeur** d'une variable est un **objet** Python

• Chaque ***objet*** appartient à une ***classe*** (ex: `int`, `str`) qui définit les *opérations* légales sur ses objets.

nombres	**booléens**	**chaînes de caractères**	**tuples**
`-234` `int` `2.34` `float`	`bool` `True` `False`	`str` `'Hello !!'`	`tuple` `(3,2,-7)`

• Les opérations sur les objets d'une classe se font à l'aide d'**opérateurs**, de **fonctions** ou de **méthodes**.

`x + 2 * z` — ↑ ↑ *opérateurs*

`x in L` — ↑ *opérateur*

`abs(x-2)` — ↑ *fonction*

`s.lower()` — ↑ *méthode de la classe* `str`

• Chaque valeur est ***typée*** (son **type** ⇔ sa **classe**). Par exemple `-234` est de type **int**, `2.34` est de type **float**, `'THX 11-38'` est de type **str**.

Ne pas confondre **affectation =** et **égalité ==**

• AFFECTATION

L'instruction `var = expr` fonctionne de la manière suivante :

a) elle calcule la valeur V de l'expression *expr*

b) elle associe à la variable *var* la valeur V obtenue.

x ← 2+3

```
>>> x = 2+3
>>> x
5
```

Exemple. L'instruction `x = 2+3` calcule 2+3, obtient 5 et à partir de maintenant x vaudra 5, jusqu'à sa prochaine affectation.

• ÉGALITÉ

L'expression $expr_1$ == $expr_2$ renvoie `True` ou `False` suivant que les valeurs des expressions $expr_1$ et $expr_2$ sont égales ou pas.

x-1 = 9-x ?

```
>>> x-1 == 9-x
True
```

NB. Chaque type de données décide ce que signifie l'égalité == pour les valeurs de ce type. Trop imprécis dans les nombres approchés (p.19) !

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- D'où vient le mot anglais "computer" pour **ordinateur** ?
- Pourquoi est-ce important de savoir programmer ?
- Donnez deux exemples de nombres de types distincts en Python.
- Combien y a-t-il d'éléments dans le type `boolean` ? Qui sont-ils ?
- Les mots type et classe sont-ils équivalents ?
- Que semble signifier le mot `def` en Python ?
- A quoi semble servir le mot `for` ?
- Quel semble être le rôle du mot `return` ?
- La distance à la marge en Python semble-t-elle être importante ?
- L'opérateur d'affectation est-il `==` ou `=` ? Donnez un exemple.
- Que signifie `x + 10 = 5` ? Même question avec `x + 10 == 5 + x`.
- Faisons `a = 2` et `b = 3`. Puis `a = b` **suivi** de `b = a`. Que **valent** a et b ?
- Comment échangeriez-vous les valeurs des variables x et y ?

Les entiers de Python (int)

• Chaque objet Python est d'un certain **type** (ou ***classe***).
On dit que -13 est de type `int` : il s'agit d'un nombre **entier relatif** ($\mathbb{Z}$).

```
>>> type(-13)
<class 'int'>
```

• **NB** : Python peut travailler avec de très grands entiers, mais dans la limite de la mémoire finie de la machine. Aucune perte de précision !

Test au toplevel :

```
>>> 5678943 * 34354556 * 67868690868757
13241016349563897766282275156
```

• Un **type** est muni d'**opérations sur les objets de ce type** : dans le type `int`, on trouve les opérations arithmétiques usuelles entre entiers :

Python :	`a + b`	`a - b`	`a * b`	`a ** b`	`a // b`	`a % b`
Maths :			ab	a^b	*quotient*	*reste*
					$a \geq 0$ et $b > 0$	

```
>>> 52 // 5
10
```
quotient

```
>>> 52 % 5
2
```
reste

Attention aux priorités des opérateurs !

+ -	* // %	**
même priorité (faible)	même priorité (haute)	(très haute)

• Lorsque deux opérateurs ont **même priorités** (ex: + et -), ils sont exécutés **de gauche à droite** :

```
a - b + c == (a - b) + c          a // b * c == (a // b) * c
```

• Sinon celui de **plus haute priorité** est exécuté en premier :

```
3 * x**2 == 3 * (x**2)
```

$3x^2$

• 5*x**2+4*y est lu comme (5*(x**2))+(4*y).

• Mettez des espaces pour la lisibilité : 5 * x**2 + 4 * y

$5x^2 + 4y$

La division euclidienne dans les entiers naturels

a // b fournit le **quotient** q de l'entier a ≥ 0 par l'entier b > 0.

a % b fournit le **reste** r de la division de l'entier a ≥ 0 par l'entier b > 0.

"modulo"

$$a == b * q + r \quad \text{avec} \quad 0 \leq r < b$$

a // b a % b

a	b
r	q

43	5
3	8

- Un entier d **divise** un entier n si la division de n par d *tombe juste*, donc lorsque son **reste est nul**, donc lorsque `n % d == 0` *(n modulo d est égal à 0)*

Essais au toplevel :

```
>>> 15 % 3
0
```

```
>>> 16 // 3
5
```

```
>>> 15 % 4
3
```

```
>>> 1234 % 10
4
```

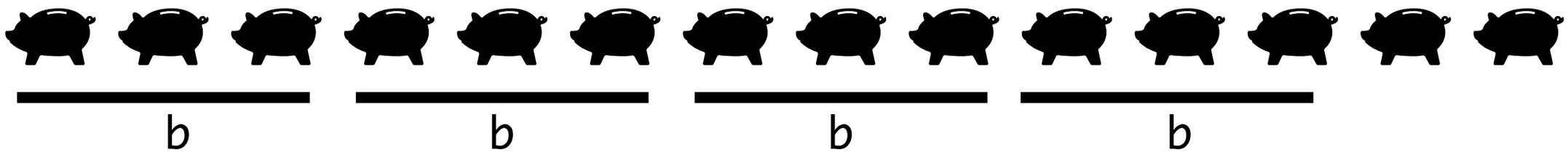
```
>>> 1234 // 10
123
```

- En particulier, l'entier n est **pair** si et seulement si `n % 2 == 0`

impair si et seulement si `n % 2 == 1`

Exemple d'algorithme : quotient et reste d'une division

• Dans 11 combien de fois 3 ? En général, dans a combien de fois b ?

a = 

• Le **quotient de a par b** est obtenu en comptant combien de fois on peut retrancher b de a. Ici on peut le retrancher 4 fois : 11 // 3 == 4

• Le **reste de la division de a par b** est ce qui reste après avoir retranché le maximum de fois b de a : 11 % 3 == 2

NB. Observez le cas particulier où a < b : on ne peut pas retrancher b.

• **ALGORITHME** du **calcul du quotient et du reste** de a ≥ 0 par b > 0.

Soit q = 0 (un compteur)

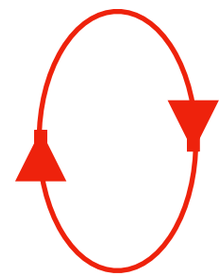
Tant que a ≥ b : *une boucle*

    a devient a - b (retrancher b de a)

    q devient q + 1 (augmenter le compteur de 1)

Le quotient est **q** est le reste est **a**

(fini !)



- Le **code Python** complet *(VERSION 1)* pourrait être :

```
#  division.py, version 1

a = 14                          #  donnée
b = 3                           #  donnée
#  Calcul du quotient q de la division de a par b
q = 0                           #  un compteur de soustractions
while a >= b :
    a = a - b                   #  retrancher b de a
    q = q + 1                   #  augmenter le compteur q
print('quotient =', q, 'et reste =', a)
```

boucle

Exécution :

```
quotient = 4 et reste = 2
```

NB. ***Indentation obligatoire*** *(distance à la marge) ! Elle exprime que les deux lignes sont "dans le while"... Le print qui suit est exécuté* ***après la fin du while****. Cette indentation est souvent automatique mais pas toujours : vérifiez-la car elle modifie la logique du programme et conduit à des erreurs sournoises !*

Quelques remarques sur ce programme

• La moindre erreur d'**indentation** est fatale : résultats faux. Avancer par tabulations, et non par espaces (2 ou 4 en général) !

• La **boucle** a pour but de modifier de manière régulière les valeurs des variables, de sorte à se rapprocher du résultat pas à pas.

• Les **commentaires** débutant par un # servent à rendre le texte plus lisible, pour soi-même plus tard, ou pour un lecteur extérieur.

• **Pourquoi** rédiger cette boucle alors qu'on dispose des opérateurs // et % ?
Juste pour s'entraîner à **comprendre, et coder** ce qu'on a compris.

***GPT** : La programmation présente un intérêt majeur pour un lycéen scientifique, car elle développe des compétences essentielles en résolution de problèmes et en logique. Elle permet d'automatiser des tâches complexes, facilitant ainsi l'analyse de données et la modélisation de phénomènes scientifiques. En apprenant à coder, les élèves acquièrent une pensée critique et une méthode de travail rigoureuse, utiles dans toutes les disciplines. De plus, la programmation ouvre des portes vers des carrières variées dans des domaines en pleine expansion, comme l'intelligence artificielle et la biotechnologie. Enfin, elle encourage la créativité en permettant de réaliser des projets innovants.*

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Est-ce que tous les **entiers relatifs** des maths sont de type `int` ?
- Est-ce que `5.0` est un entier en Python ?
- L'opération `a / b` retourne-t-elle le quotient de a par b dans le type `int` ?
- Quelle est l'opération permettant de savoir si n est multiple de 3 ?
- Combien vaut `1000 // 5 % 3` ?
- Combien vaut `3 * 5 // 2 - 2 ** 2 * 3` ?
- Quel est l'opérateur le plus prioritaire parmi `%`, `//` et `*` ?
- Que signifie le mot `def` en Python ?
- Quand dit-on qu'un texte de programme est mal indenté ?
- L'indentation d'un programme est-elle obligatoire ou esthétique ?
- Si a,b,q,r sont entiers avec a = bq+r, alors q est le quotient de a par b. Exact ?...
- Quel est l'intérêt d'utiliser une boucle en programmation ?
- Où est la touche de tabulation sur un clavier d'ordinateur ?
- Qui est GPT ? Avez-vous déjà dialogué avec lui ?...

Version 2 : coder une FONCTION

• Plutôt que rédiger le cas particulier de a=11 et b=3, mieux vaut **faire abstraction** des valeurs des données a et b, et programmer l'algorithme portant sur ces variables avec une **fonction**.

```
#  division.py, version 2

def aff_division(a,b) :    #  affichage du quotient et du reste
    #  on suppose a et b entiers, a ≥ 0 et b > 0
    q = 0
    while a >= b :              #  tant que l'on peut retrancher b
        a = a - b
        q = q + 1
    print(a, '=', q, '*', b, '+', a)

aff_division(14,3)
aff_division(1024,4)
```

Exécution :

```
14 = 4 * 3 + 2
1024 = 256 * 4 + 0
```

*NB. Cette fonction **fait** quelque chose, elle n'a ici **pas de résultat** contrairement aux maths.*

Eléments du langage Python : les **boucles while et for**

• Une **boucle** a pour but de modifier de manière répétitive les valeurs de certaines variables (dites *variables de boucle*), de sorte à se rapprocher du résultat pas à pas.

• Si l'on ne connaît pas à l'avance le nombre de tours de boucle, on utilise une boucle `while`. Il faut alors disposer d'une **condition de sortie**. Exemple : la division, avec la condition de sortie `a < b`.

• Assez souvent, la boucle est gouvernée par une variable décrivant un **intervalle fini** d'entiers. On peut toujours utiliser un **while**, mais la boucle **for** :

```
for … in range(a,b) :
```

s'avère plus adaptée.

Exemple. Affichage de la somme des entiers de 1 à n inclus. La donnée est n ≥ 0 entier, qui devient le **paramètre** d'une fonction.

```
def aff_somme(n) :     # affiche 1+2+...+n
    s = 0
    for k in range(1, n + 1) :
        s = s + k
    print('1 + … +',n,' =',s)
```

Test au toplevel :

```
>>> aff_somme(10)
 1 + … + 10  = 55
```

• La fonction **range** de Python produit les entiers d'un intervalle, ils seront *générés* un à un par la boucle for. Trois modes d'utilisation :

```
range(a)        #  les entiers de [0 ; a]
range(a,b)      #  les entiers de [a ; b-1]
range(a,b,s)    #  les entiers de [a ; b-1] espacés de s
```

• Somme res des carrés des entiers de [0 ; 10] :

```
res = 0
for i in range(11) :
    res = res + i * i
```

• Somme des carrés des entiers de [10 ; 20] :

```
res = 0
for k in range(10, 21) :
    res = res + k * k
```

• Somme des carrés des entiers **impairs** de [10 ; 20] :

```
res = 0
for j in range(11, 21, 2) :
    res = res + j * j
```

ATTENTION : a, b, s doivent être des entiers !

Les **fonctions** qui retournent **un résultat**

• Les fonctions `aff_division` et `aff_somme` ont un **effet**, celui d'afficher du texte. Mais aucun résultat (leur résultat est None qui ne s'affiche pas).

```
>>> aff_somme(10) == None
1 + … + 10  = 55                # l'effet de aff_somme
True                            # mais son résultat est None
```

• Une autre (meilleure) façon de coder consiste à programmer des **fonctions retournant un (seul) résultat**, qui sera affiché en-dehors de la fonction. Cela permet en effet de COMPOSER les fonctions !

```
def somme(n) :      # renvoie le résultat de 1+2+...+n
    s = 0
    for k in range(1, n + 1) :
        s = s + k
    return s        # le résultat est s

print('1 + … +10 =', somme(10))
```

Exécution :

```
1 + … +10 = 55
```

Le type tuple (couples, triplets, etc.)

• Les couples, triplets,… des maths se nomment en Python des **tuples**, dans la classe `tuple`, et se notent (x,y), (x,y,z),… La longueur d'un tuple `t` s'obtient par `len(t)` et ses composantes se notent `t[i]`, avec $i \geq 0$.

```
>>> t = (5, 7+2, -2)
>>> t
(5, 9, -2)
```

```
>>> t[0]
5
>>> (t[2], t[1])
(-2, 9)
```

```
>>> len(t)
3
```

• Une **fonction à plusieurs résultats** renverra un **tuple** de résultats !

```
def division(a,b) :      #  cf. division de a par b, page 17
    q = 0
    while a >= b :
        a = a - b
        q = q + 1
    return (q,a)
```

```
d = division(14,3)
q = d[0]
r = d[1]
```

NON

```
(q,r) = division(14,3)
```

Le **domaine** d'une fonction

• On ne code que la manière de calculer : le **domaine où elle calcule** est sous-entendu par le programmeur. La fonction calcule ce qu'elle peut !

```
def f(a,b):
    return 2*a + b
```

```
>>> f(3,4)
10
```

```
>>> f(3,0.2)
6.2
```

```
>>> f('+','un')
'++un'
```

```
>>> f(3,'un')
TypeError
```

• **Renforcer la sécurité** : on précise en commentaire les **types** des paramètres et ce qu'elle calcule. C'est à l'utilisateur de la fonction de respecter les types des paramètres !

```
def f(a,b):      #  a,b sont des entiers naturels, etc.
    ...
```

un simple commentaire

```
def f(a,b):
    '''a et b sont des entiers naturels, etc.'''
    ...
```

mieux : une docstring

Test au toplevel :

```
>>> help(f)
a et b sont des entiers naturels, etc.
```

aide en ligne !

Exprimer des **conditions** dans un calcul : if...else...

• Très souvent, le programmeur va piloter le calcul en interrogeant les données. Suivant les valeurs des variables, il poursuivra le calcul de telle ou telle manière. Exemple typique : la valeur absolue abs(n).

```
def valabs(n) :    # n est un nombre
    if n >= 0 :
        return n
    else :
        return -n
```

Test au toplevel :

```
>>> valabs(-5)
5
```

• Jet d'un **dé équilibré à 6 faces**. On importe la fonction randint(a,b) du module random qui contient des utilitaires sur le hasard. Cette fonction renvoie un entier **aléatoire** de [a ; b].

```
from random import randint

def jet() :    # sans paramètre
    return randint(1,6)
```

```
>>> randint(-50,50)
-12
>>> (jet(), jet())
(4, 1)
```

• Programmons une pièce de monnaie truquée à deux faces qui va renvoyer 'pile' avec 1 chance sur 3. Je renvoie un *texte* (en Python, on parle de **chaîne de caractères**, ou simplement **chaîne**) `'pile'` ou `'face'`.

```
def jet13() :
    hasard = randint(1,3)          #  1,2 ou 3 au hasard
    if hasard == 1 :
        return 'pile'              #  1 chance sur 3
    else :
        return 'face'              #  2 chances sur 3

for i in range(10) :
    print(jet13(), end=' ')        #  ne pas aller à la ligne
print()                            #  aller à la ligne à la fin
```

Exécution :

```
pile face pile face face face pile pile face face
```

• Le mot **aléatoire** signifie *au hasard*. C'est un hasard calculé par des formules mathématiques : pas le vrai hasard, mais presque…

• S'il y a plus de 2 cas, on utilise `if...elif...else...`, où `elif` est une contraction de `else if`. Programmons un **dé équilibré** à 3 faces renvoyant une chaîne `'bleu'`, `'blanc'` ou `'rouge'` au hasard.

```
def bbr() :
    hasard = randint(1,3)      #  1,2 ou 3
    if hasard == 1 :
        return 'bleu'
    elif hasard == 2 :
        return 'blanc'
    else :
        return 'rouge'
```

Test au toplevel :

```
>>> (bbr(), bbr(), bbr())
(rouge, rouge, bleu)
```

• NB1 : Le mot `return` provoque une *sortie **immédiate** de la fonction* avec le résultat sous le bras. Donc ici on pourrait écrire en plus concis :

```
def bbr() :
    hasard = randint(1,3)      #  1,2 ou 3
    if hasard == 1 : return 'bleu'
    if hasard == 2 : return 'blanc'
    return 'rouge'
```

mais plus dangereux, attention !...

Les variables locales d'une fonction

• Vous avez noté que la variable hasard ci-dessus (ci-dessous à gauche) était introduite **dans** la fonction : elle est dite **variable locale de (à) la fonction**. Elle n'est pas utilisable à l'extérieur du texte de la fonction.

• Si hasard avait été défini **à l'extérieur** de la fonction, le résultat de la fonction jet13 aurait toujours été le même ! La variable hasard aurait été **globale** et utilisable (mais non modifiable*) dans d'autres fonctions.

• Les variables qui sont locales aux fonctions sont largement **préférées** par les programmeurs aux variables globales. A bon entendeur…

```
def jet() :
    hasard = randint(1,6)
    return hasard
```

Il y a ici deux variables 'hasard', l'une est locale à la fonction jet et l'autre est globale.

Test au toplevel :

```
>>> hasard = 10
>>> (jet(), jet(), jet())
(5, 1, 3)
>>> hasard
10
```

(*) Il existe un mot-clé global *permettant de modifier une variable globale. À éviter...*

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Pourquoi les fonctions sont-elles importantes en programmation ?
- Quel est l'effet de la fonction `aff_division` ? Quel est son résultat ?
- Quand doit-on opter pour une **boucle** `for` et non une **boucle** `while` ?
- Programmez une fonction `nbchiffres(n)` qui renvoie le nombre de chiffres de l'entier n > 0 ? Utilisez-vous `while` ou `for` et pourquoi ?
- Idem pour la fonction `prem_chif(n)` qui renvoie le premier chiffre de n.

Exemple : `prem_chif(5367831) == 5`

- Voulez-vous bien coder le calcul de la somme 0.5 + 1 + 1.5 + 2 +…+ 99.5 + 100 ?
- Quelle est l'instruction permettant d'utiliser la fonction `randint` ?
- Utilisez la fonction `jet13`. Vérifiez que sur 10000 tirages, il y a bien environ une chance sur trois d'obtenir `'pile'`.
- Comment indiquez-vous des contraintes sur les arguments d'une fonction ?
- Quand dit-on qu'une variable est locale à une fonction ?
- Est-il sain de vouloir modifier une variable globale dans une fonction ?

Quelques algorithmes sur l'arithmétique des entiers naturels

- Nous allons travailler dans le type `int`, avec des entiers tous positifs.

- L'arithmétique des entiers est basée sur la notion de **divisibilité**. En Python, on traduira ***p divise q*** par : `p % q == 0` où `%` se lit *modulo*.

- Combien un entier n admet-il de diviseurs, exceptés 1 et n ?

NB. Les *vrais* diviseurs 1 < d < n sont nommés ***diviseurs propres*** de n.

```
def nbdiv(n) :          # renvoie le nombre de diviseurs propres de n
    res = 0
    for k in range(2,n) :          # pour k = 2..n-1
        if n % k == 0 :            # si k divise n
            res = res + 1          # on le compte (pas de else)
    return res                     # le résultat

print('1400 admet',nbdiv(1400),'diviseurs propres')
```

```
1400 admet 22 diviseurs propres
```

• Il y a 22 diviseurs, mais peut-on les **afficher** ?

```
def affdiv(n) :    # afficher les diviseurs propres de n
    for k in range(2,n) :          # pour k = 2..n-1
        if n % k == 0 :            # si k divise n
            print(k,end=' ')       # à l'horizontale...
    print()                        # aller à la ligne

print('Les diviseurs propres de 1400 sont :')
affdiv(1400)
```

Exécution :

```
Les diviseurs propres de 1400 sont :
2 4 5 7 8 10 14 20 25 28 35 40 50 56 70 100 140 175 200 280 350 700
```

• Vous avez noté que la partie `else` est **optionnelle**. Si on n'a rien à faire, on ne fait rien, on passe son tour (c'est l'instruction `pass` de Python).

• Il n'y a pas de `return` : la fonction ne renvoie **aucun résultat**, elle se contente de faire un **effet**, ici un affichage.

• Un entier n est dit **premier** s'il est ≥ 2 et s'il n'admet **aucun diviseur propre** (distinct de 1 et n) : il n'est **pas décomposable, irréductible** !

• Les entiers ≥ 2 non premiers sont dits **composés** (de nombres premiers).

```
def est_premier(n) :    # l'entier n est-il premier ?
    return (n ≥ 2) and (nbdiv(n) == 0)

print("Les nombres premiers jusqu'à 100 :")
for k in range(2,101) :
    if est_premier(k) :
        print(k,end=' ')
print()                 # pour aller à la ligne à la fin
```

```
Les nombres premiers jusqu'à 100 :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

• L'opération `a and b` prend des booléens a, b valant `True` ou `False`, et retourne `False` sauf si a et b sont vrais tous les deux.

• Idem avec `a or b` qui retourne `True` sauf si a et b sont faux tous les deux. La négation se note `not` : `if not est_premier(n) : ...` et la non égalité se note `!=`

La fonction est_premier est **correcte** mais le programmeur se pose aussi le problème de **l'efficacité**. Son programme peut-il être **accéléré**, la **mémoire** de l'ordinateur est-elle suffisante pour l'exécuter ?

• Ici, l'utilisation de nbdiv est simple mais naïve. Elle nécessite de connaître **tous** les diviseurs propres de n alors qu'il suffit de savoir qu'il y en a **au moins un**, ce qui est beaucoup plus rapide ! Cherchons ***le plus petit diviseur de n qui soit*** ≥ 2. Dans le pire des cas, ce sera n lui-même et n sera premier.

```
def ppdiv(n):
    '''Retourne le plus petit diviseur ≥ 2 de l'entier n ≥ 2'''
    if n % 2 == 0:                  # cas particulier : n est pair
        return 2                    # on s'échappe avec le résultat 2
    k = 3                           # (sinon) soit k=3 le premier candidat
    while n % k != 0:               #  tant que k ne divise pas n :
        k = k + 2                   #         on passe à l'impair suivant
    return k                        #  le résultat sera n dans pire des cas
```

Tests au toplevel :

```
>>> ppdiv(45)
3       # premier
```

```
>>> ppdiv(47)
47      # premier
```

```
>>> ppdiv(431941)
61      # premier
```

• La fonction `est_premier` est aussitôt modifiée et devient beaucoup plus rapide lorsque n n'est pas premier.

```
def est_premier(n) :     # l'entier n est-il premier ?
    return (n >= 2) and (ppdiv(n) == n)
```

Tests au toplevel :

```
>>> est_premier(61)
True        # premier
```

```
>>> a = randint(2,1000)
>>> b = randint(2,1000)
>>> est_premier(a * b)
False        # c'était prévisible !
```

• Le **calcul** de `est_premier(n)` consomme environ n divisions dans le pire des cas pour faire son travail. Qui sait qu'il n'y a pas **mieux** ? Il y a souvent mieux, mais il faut de plus en plus de neurones… Allez, essayons :

• Observons que **les diviseurs sont groupés par paires** : si d divise n, alors l'entier D = n // d est aussi un diviseur de n. Ah, mais si d ≤ $\sqrt{n}$ alors D ≥ $\sqrt{n}$ (ok ?). Donc si l'on ne trouve rien jusqu'à $\sqrt{n}$, c'est qu'il n'y en aura pas d'autre que n. Je vous laisse les détails du nouveau programme qui consommera au pire $\sqrt{n}$ divisions ce qui est beaucoup plus bas que n…

LA DECOMPOSITION EN FACTEURS PREMIERS

• Nous sommes partis du bas de la montagne. Courage, grimpons…

• **Tout entier n ≥ 2 se décompose de manière unique en produit de nombres premiers**. Il s'agit de trouver à partir de la donnée n les facteurs premiers et l'exposant de chacun.

$$13 = 13^1$$

*13 est **premier***

$$27744 = 2^5 \times 3^1 \times 17^2$$

*27744 est **composé***

• Le problème général n'a toujours pas de solution en 2024 pour les très grands nombres premiers (des milliers de chiffres). Nous ne traiterons que des **petits** nombres premiers, disons jusqu'au milliard…

Et c'est bien car les codes secrets resteront secrets. Ils sont pour la plupart basés sur l'arithmétique des nombres premiers. Chut, ne le répétez pas…

• Notre fonction de base **exposant(p,n)** va retourner **l'exposant du nombre premier p dans la décomposition de n**. Je vous la laisse…

```
>>> exposant(17, 27744)
2
```

```
>>> exposant(5, 27744)
0
```

• Nous ne disposons pas de liste de nombres premiers. Pour décomposer $n \geq 2$, nous allons donc parcourir les entiers ≥ 2 et pour chaque nombre premier calculer s'il figure dans la décomposition. Au fur et à mesure, le nombre n va diminuer jusqu'à ce qu'il vaille 1.

```
def affdecomp(n) :          # affichage de la décomposition de n ≥ 2
    p = 2                   # le premier candidat
    while n > 1 :
        e = exposant(p,n)  # n = p^e x m
        if e > 0 :
            n = n // p**e   # n = m diminue
            print(p,'**',e,sep='',end=' ')
        p = p + 1           # prochain candidat
    print()
```

Pourquoi p est-il premier ?

Tests au toplevel :

```
>>> affdecomp(27744)
2**5 3**1 17**2
```

```
>>> affdecomp(13)
13**1
```

• `sep=''` exprime qu'il y a une **sép**aration **vide** entre deux affichages.

• En traitant p=2 à part, on pouvait aller de 2 en 2 à partir de p=3.

LE PLUS GRAND COMMUN DIVISEUR : PGCD

• Le PGCD des entiers a et b > 0 est leur Plus Grand Commun Diviseur. En python, c'estla fonction gcd (Greatest Common Divisor) du **module** math.

Tests au toplevel :

```
>>> from math import gcd
>>> gcd(8,12)
4
```

```
>>> gcd(8,35)
1
```

*8 et 35 sont **premiers entre eux***

• Le PGCD peut se calculer à partir des décompositions de a et b, en conservant les facteurs premiers en commun avec leur plus petit exposant.

```
>>> affdecomp(gcd(2**3 * 5 * 7**2, 2**4 * 3 * 5**2))
2**3 5**1
```

• Lorsque le PGCD vaut 1, les deux nombres sont dits **premiers entre eux**. Ce qui signifie qu'ils n'ont **pas de facteur premier en commun** dans leurs décompositions. Par exemple 8 = 2^3 et 35 = 5 x 7.

• Mais Python ne passe **pas** par les décompositions pour le programmer !...

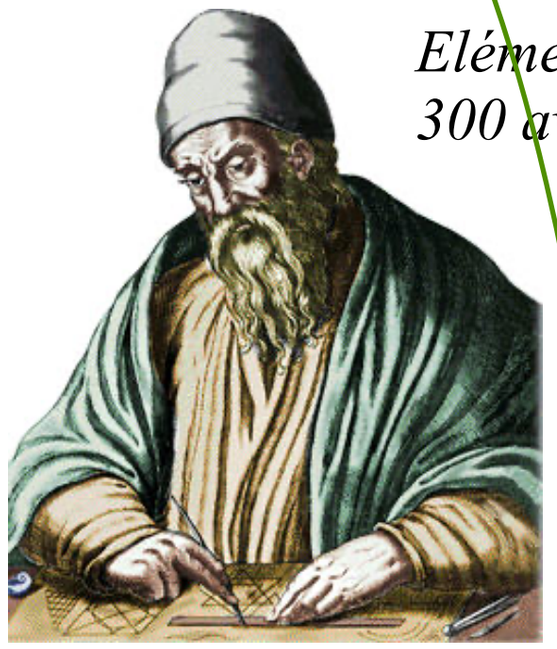
• Pour calculer un PGCD, Python utilise **l'algorithme d'Euclide**, l'un des premiers grands algorithmes de l'humanité !

• Soit à calculer PGCD(a,b) avec a et b entiers naturels.

EUCLIDE :

```
PGCD(a,0) = a
PGCD(a,b) = pgcd(b,a % b) si b > 0
```

```
PGCD(8,12) = PGCD(12,8 % 12)
           = PGCD(12,8)
           = PGCD(8,12 % 8)
           = PGCD(8,4)
           = PGCD(4,8 % 4)
           = PGCD(4,0)
           = 4
```



Eléments, Livre VII
300 av. JC

```
def pgcd(a,b) :
    while b > 0 :
        ???????
    return a
```

à compléter…

```
>>> pgcd(8,12)
4
```

```
def pgcd(a,b) :
    if b == 0 : return a
    return pgcd(b,a % b)
```

Oui, possible.
En Première…

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Comment se lit à haute et intelligible voix l'opération `a % b == 0` ?
- Quel est le nombre de diviseurs de n=20 ?
- Que signifie diviseur propre d'un entier n ?
- Quel est le résultat de la fonction `affdiv(n)` ?
- Dans l'instruction `if`… la partie `else` est-elle obligatoire ?
- Modifiez `affdiv(n)` en `affdivimp(n)` qui n'affichera que les diviseurs impairs de l'entier n.
- Le résultat de `ppdiv(n)` est-il toujours premier ? Pourquoi ?
- Le nombre de diviseurs de n peut-il se déduire (à la main) de sa décomposition en facteurs premiers ?
- Rédigez le programme utilisant $\sqrt{n}$ cité au bas de la page 32.
- Améliorez `affdecomp(n)` en traitant p=2 à part (page 34).
- Complétez la fonction `pgcd` en bas à gauche de la page 36.
- Vérifiez que la version à droite fonctionne. Cela vous choque-t-il ?

Les nombres approchés de Python (float)

• Comme 3.14159. Dites **nombre flottant** plutôt que *nombre réel*. Les nombres réels sont mathématiquement bien plus compliqués !

• Les nombres flottants sont des approximations (une quinzaine de chiffres après le point décimal). **Le calcul flottant est approché !**

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 0.5
True
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

• Cet exemple doit vous dégoûter d'utiliser l'égalité sur des nombres flottants ! Fixez-vous une précision, par ex. h = 0.01, et demander si |a - b| < h plutôt que a == b. Simple conseil d'ami…

• Les fonctions de l'Analyse comme le sinus, $\sqrt{}$ ou même le nombre π, etc. doivent être **importées** à partir du **module** `math` (comme gcd en p. 35).

• La **notation scientifique** 1e-3 en Python signifie 10^{-3}, soit 0.001. De même 3.14e3 signifie 3,14 x 10^3 soit 3140.0 (c'est un flottant).

• Exemple : **tableau de points** d'une fonction numérique

• J'étudie la fonction $f : x \mapsto \dfrac{1}{1+x^2}$ définie sur $\mathbb{R}$.

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

• Pour la dessiner point par point sur [-1 ; 1], j'ai besoin de certains points sur la courbe, espacés de 0.2 :

```
x = -1
while x <= 1:
    print(x,f(x))
    x = x + 0.2
```

```
-1 0.5
-0.8 0.6097560975609756
-0.6000000000000001 0.7352941176470588
-0.4000000000000001 0.8620689655172413
-0.2000000000000007 0.9615384615384615
-5.551115123125783e-17 1.0
0.1999999999999996 0.9615384615384615
0.3999999999999997 0.8620689655172414
0.6 0.7352941176470589
0.8 0.6097560975609756
1.0 0.5
>>>
```

*NB. Nous sommes dans le **calcul approché**... Mais pour n'afficher y qu'avec 2 chiffres après le point décimal, utilisez* `round(y,2)`.

• RAPPEL : Une **variable locale à une fonction** permet de **donner un nom à un calcul intermédiaire**, de manière à décomposer les calculs pour obtenir un texte plus lisible.

• Souvent, elle permettra de **ne pas calculer plusieurs fois la même quantité** (le temps c'est de l'argent !). Exemple avec les fonctions :

$$f : x \mapsto \frac{x^2}{1 + x^2}$$

```
def f(x):
    x2 = x*x                    #  x² n'est pas calculé deux fois !
    return x2 / (1 + x2)
```

$$g : x \mapsto \frac{\sin x^2}{x^2 + \sin x^2}$$

```
def g(x):
    x2 = x*x                    #  x² n'est pas calculé deux fois !
    sx2 = sin(x2)               #  idem pour sin(x²)
   return sx2 / (x2 + sx2)
```

*Oui, oui, je sais, il faut **importer** la fonction **sin** qui est dans le **module math**…*

Qu'est-ce qu'un **module** en Python ?

• Lorsque vous lancez le logiciel Python, vous obtenez un langage ayant un grand nombre de **fonctions** (`print`, `abs`, `max`, `range`,…), d'**opérateurs** (+, -, *,…), de **classes** d'objets (`int`, `float`,…), de **mots-clés** (`def`, `if`,…).

• Pour un travail plus spécialisé (maths, dessin, web, musique,…), Python dispose de **modules** : des bibliothèques de fonctions et classes qu'il faudra **importer** pour pouvoir les utiliser.

• Le module **math** sera notre ami. Il contient les fonctions usuelles des maths comme la racine carrée (`sqrt`), le PGCD (`gcd`), le nombre `pi`, etc.

```
>>> from math import sqrt, pi
>>> sqrt(pi)
1.7724538509055159      # √π
```

ou

```
>>> import math
>>> math.sqrt(math.pi)
1.4142135623730951
```

• Vous voulez des fractions ?
Utilisez le module `fractions` :

Google : "fractions en Python"

```
>>> from fractions import Fraction
>>> Fraction(1,2) + Fraction(2,3)
Fraction(7,6)
```

Le module random

• Nous avons déjà en page 23 rencontré le module random contenant des fonctions permettant de coder des problèmes *stochastiques (avec du **hasard**)*.

- randint(a,b) prend deux **entiers** a et b avec a ≤ b, et retourne un **entier** tiré au hasard dans [a ; b]. Le **dé à 6 faces** est randint(1,6) .

```
from random import randint
for i in range(10):
    print(randint(1,6),end=' ')
```

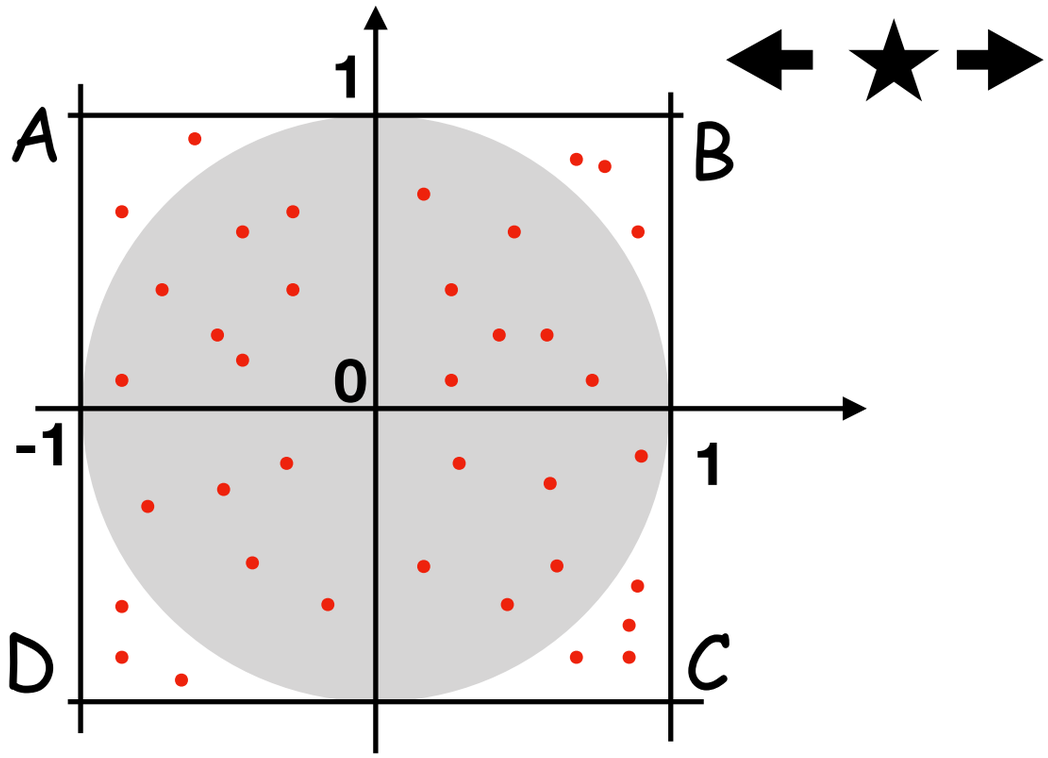
```
6 3 6 1 3 2 6 4 5 5
```

- uniform(a,b) prend deux **flottants** a et b avec a ≤ b, et retourne un **flottant** tiré au hasard dans]a ; b[.

```
from random import uniform
for i in range(5):
    print(uniform(-1,1))
```

```
0.3257240119576858
-0.1106933161769800
0.4659100966112690
-0.2661807946190950
-0.5934745200942750
```

Lancer aléatoire de fléchettes



• Il s'agit d'une *expérience de Monte-Carlo*, consistant à **faire un calcul en utilisant le hasard**. Exemple : calculer l'aire d'un disque de rayon 1 (on doit trouver π).

• Idée : lancer n fléchettes dans le carré ABCD et compter le nombre s (succès) de celles qui atterissent dans le disque gris. La probabilité de tomber dans le disque s'obtient de deux façons : s/n, ou bien le rapport aire(disque)/aire(carré) qui vaut π/4. D'où la valeur approchée π = 4s/n avec n grand.

• Pour vérifier si la fléchette, un point (x,y) choisi au hasard avec uniform, est dans le disque, on vérifie si sa distance à O est < 1.

```
from math import sqrt

def dist0(x,y) :    # distance à l'origine
    return sqrt(x**2 + y**2)
```

```python
def monte_carlo(n) :
    '''calcul de π par lancer de n fléchettes sur une cible de rayon 1'''
    s = 0           #  le nombre de succès (lancers arrivant dans le disque)
    for k in range(n) :          #  n fois
        x = uniform(-1,1)        #  je tire au hasard(x,y) dans le carré
        y = uniform(-1,1)
        if dist0(x,y) < 1 :      #  si (x,y) tombe dans le disque,
            s = s + 1            #     succès !
    print(s,'fléchettes dans le disque, sur un total de',n)
    return(4 * s / n)
```

```
>>> monte_carlo(1000)
771 fléchettes dans le disque, sur un total de 1000
3.084
```

effet :

résultat :

```
pi = 3.141592653589793
```

```
>>> monte_carlo(10000)
7867 fléchettes dans le disque, sur un total de 10000
3.1468
```

• L'approximation n'est ici pas extraordinaire mais le principe reste intéressant pour calculer en l'absence de formules exactes…

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Est-ce que `0.3 - 0.2 == 0.1` en Python ?
- Est-ce raisonnable d'utiliser `==` avec des nombres flottants ?
- De quel type est le résultat de l'addition d'un entier et d'un flottant ?
- Si x est un flottant, que renvoie `int(x)` si $x \geq 0$ ou $x < 0$?
- Si x est un entier, que renvoie `float(x)` ?
- Vérifiez expérimentalement à l'aide d'une boucle que la quantité $(1 - \cos x)/x^2$ peut être rendue à une distance de 0.5 inférieure à `0.001` à condition que x soit suffisamment proche de 0 (à quelle distance ?).
- Si j'exécute `import math`, quel est le type de la variable math ?
- Montrez comment vous utiliseriez les générateurs de nombres aléatoires, dans les entiers puis dans les flottants.
- En utilisant le module `fractions`, calculez le dénominateur du nombre rationnel 1/2 + 1/3 + 1/4 + … 1/100. Combien a-t-il de chiffres ?
- Quelle est la probabilité que deux entiers tirés au hasard dans [2,100000] soient premiers entre eux (page 35) ? Est-ce proche de $6/\pi^2$?…

Les vecteurs du plan

• Nous travaillons dans un plan muni d'un repère orthonormé. Un point et un vecteur seront donc représentés par un couple (x,y). Les fonctions préciseront si (x,y) représente un point M ou un vecteur $\vec{u}$.

• Le **point** de coordonnées (x,y)

```
def point(x,y) :
    return (x,y)
```

• Le **vecteur** de coordonnées (x,y)

```
def vecteur(x,y) :
    return (x,y)
```

• Les fonctions **coordonnées** d'un point-ou-vecteur pv :

```
def xpv(pv) :    # abscisse
    return pv[0]
```

```
def ypv(pv) :    # ordonnée
    return pv[1]
```

• Nous aurons besoin de construire un **vecteur** $\overrightarrow{AB}$.

```
def pvecteur(A,B) :      # vecteur d'origine A et d'extrémité B
    return vecteur(xpv(B) - xpv(A), ypv(B) - ypv(A))
```

```
>>> M = point(0,1)
>>> N = point(3,4)
```

```
>>> pvecteur(M,N)
(3,3)
```

```
>>> milieu(M,N)
(1.5,2.5)
```

?

à programmer

Opérations sur les vecteurs du plan

- On peut les **additionner**, les **soustraire**, les **multiplier par un nombre**.

```
def vadd(u,v) :              # u et v vecteurs
    return u + v
```

NON *:*
(1,2) + (3,4) == (1,2,3,4)

```
def vadd(u,v) :              # u et v vecteurs
    return (u[0] + v[0], u[1] + v[1])
```

Trop dépendant du choix de tuples pour représenter un vecteur

```
def vadd(u,v) :              # u et v vecteurs
    return (xpv(u) + xpv(v), ypv(u) + ypv(v))
```

On construit un point ou un vecteur ?

```
def vadd(u,v) :              # u et v vecteurs
    return vecteur(xpv(u) + xpv(v), ypv(u) + ypv(v))
```

Acceptable (idem pour `vsub`*)*

```
def vmul(k,u) :              # k nombre et u vecteur
    return vecteur(k * xpv(u), k * ypv(u))
```

```
>>> u1 = vecteur(3,-4)
>>> u2 = vecteur(1,6)
```

```
>>> vadd(u1,u2)
(4,2)
```

```
>>> vmul(-2,u1)
(-6,8)
```

• Travaillant avec des ***nombres flottants***, nous ne pouvons PAS utiliser le signe == pour tester l'égalité. Python fournit **isclose(a,b)** pour tester si deux nombres flottants "usuels" sont assez proches…

```
def egaux(pv1,pv2) :
    # ces vecteurs ou points sont-ils "égaux" comme flottants usuels ?
    return isclose(xpv(pv1),xpv(pv2)) \
           and isclose(ypv(pv1),ypv(pv2))    #  tolérance ok...
```

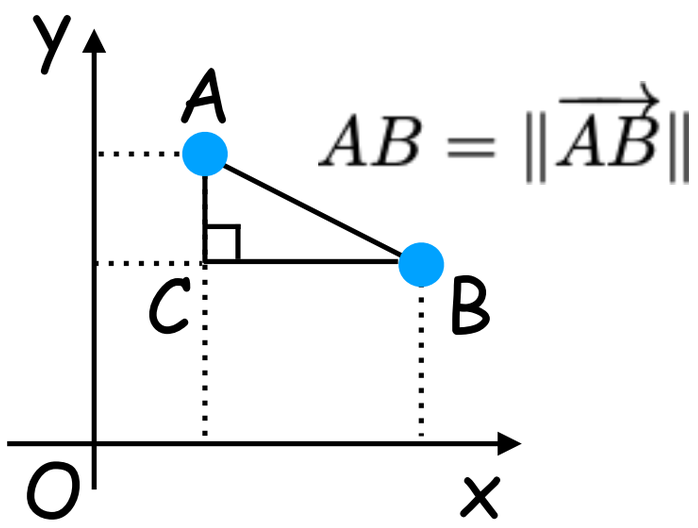
Notez le \ pour exprimer que la ligne se poursuit…

• La **distance** de deux points A et B se calcule avec le théorème de Pythagore $AB^2 = AC^2 + CB^2$ ou bien avec la norme du vecteur $\overrightarrow{AB}$.

```
def distance(A,B) :
    return norme(pvecteur(A,B))
```

? *en exo p. 50*

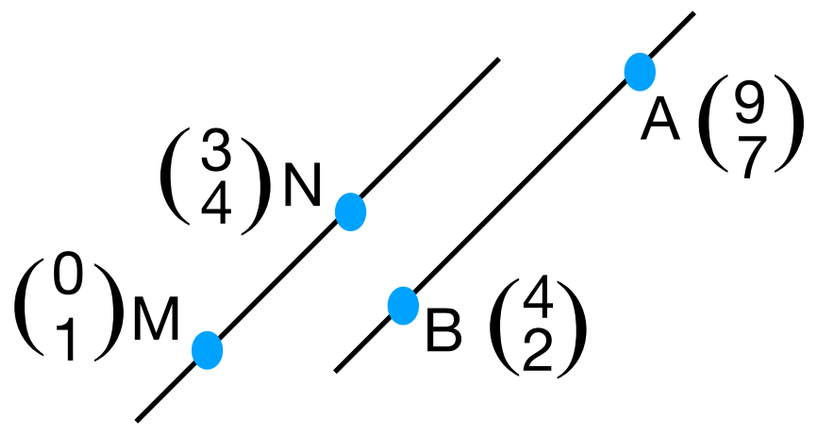
```
>>> distance(point(0,0),point(1,1))
1.4142135623730951          # √2
```



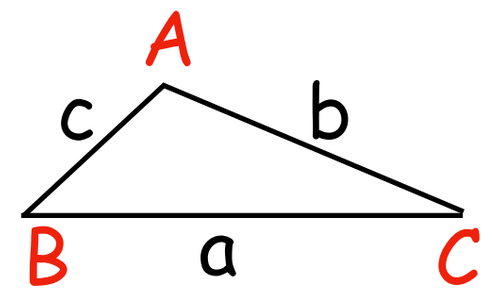
1) Soient A,B,M,N des points du plan. On peut vérifier si la droite AB est parallèle à la droite MN par un **déterminant**.

EXOS

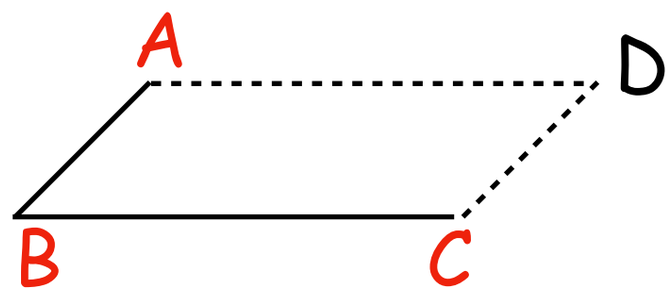
```
>>> sont_parallèles(A,B,M,N)
True
>>> sont_parallèles(A,M,N,B)
False
```



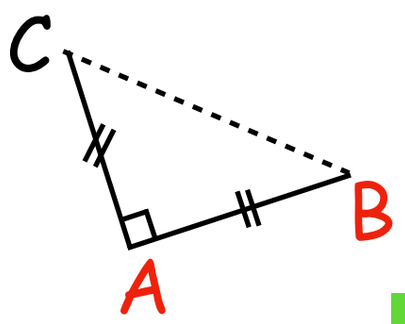
2) Soient A,B,C trois points du plan non alignés. Programmez une fonction `aire(A,B,C)` renvoyant l'aire du triangle ABC (vous calculerez les longueurs a,b,c). Cherchez sur le Web la **formule de Héron**. Testez votre fonction en des points A,B,C choisis au hasard.



3) Programmez une fonction `para(A,B,C)` renvoyant <u>le</u> point D tel que ABCD soit un parallélogramme.



4) Programmez une fonction `rect(A,B)` renvoyant <u>un</u> point C tel que ABC soit un triangle rectangle isocèle de sommet A.



POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Programmez la fonction `milieu(A,B)` renvoyant le point milieu du segment AB.
- Programmez la fonction `norme(u)` renvoyant la norme du vecteur u. Testez alors la fonction `distance(A,B)` de la page 48.
- Programmez la fonction `prodscal(u,v)` renvoyant le produit scalaire des vecteurs u et v. Puis redéfinissez `norme` à l'aide de `prodscal`.
- Programmez une fonction `unitaire(u)` prenant un vecteur u non nul et renvoyant le vecteur unitaire de même direction et sens que u.
- Programmez la fonction `ortho(u,v)` retournant `True` ou `False` suivant que les vecteurs u et v sont orthogonaux ou pas.
- Codez la fonction de translation `trans(M,v)` prenant un point M et un vecteur v, et renvoyant le point N tel que $\overrightarrow{MN} = \vec{v}$.
- Faites tous les exercices de la page précédente, inventez-en un autre !
- *Optionnel* : Pourriez-vous regrouper `vecteur` et `pvecteur` en une seule fonction utilisant un test des paramètres ?

Le module turtle

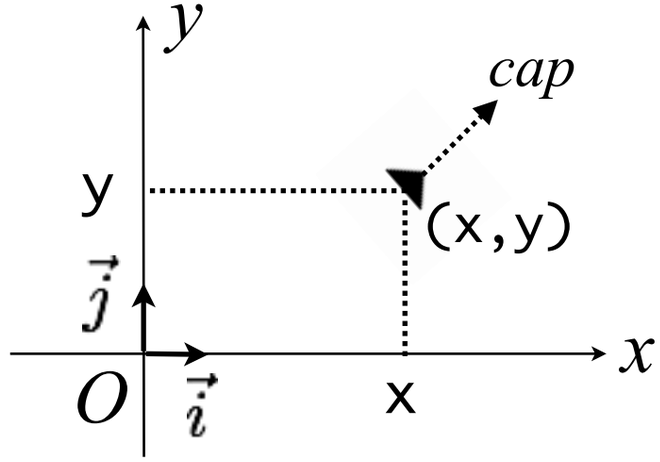
```
from turtle import *
```

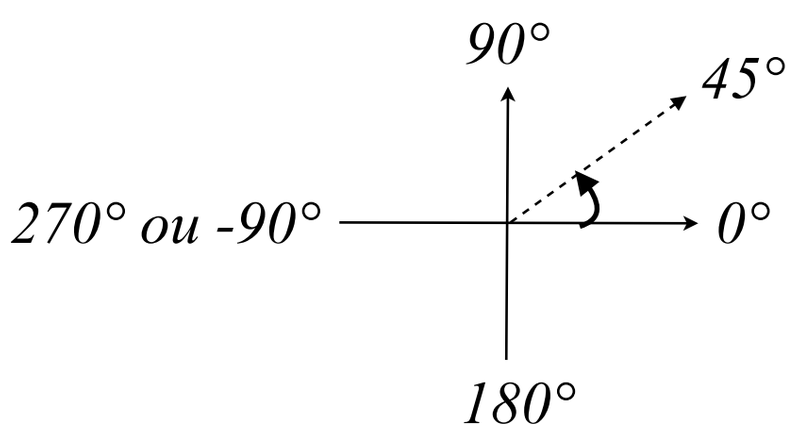
• Le programmeur pilote une tortue qui tient un **crayon** (*pen*). Elle est située à une **position** (x,y) dans un repère $(0, \vec{i}, \vec{j})$ orthonormé. L'origine 0 est au **centre** de la fenêtre graphique dont les dimensions sont par ex. 600 x 600 (pixels). Le sens positif des angles est celui des maths.

• La tortue a aussi un **cap** (*heading*) de 0° à 359°. Elle **avance** toujours en direction de son cap, et peut **tourner sur place** pour changer de cap.

tortue

position: (90,60)
cap: 45°
crayon: baissé
image: arrow





- Au départ (en mode `'standard'`) elle est au centre, cap Est (0°), crayon baissé prête à tracer, c'est le mode par défaut. Si vous adoptez le mode `'logo'`, le cap 0° est au Nord. La fenêtre graphique fait 400x300.

```
shape = 'classic'
mode = 'standard'
```

https://docs.python.org/3/library/turtle.html

• La tortue a une **image** (*shape*) à l'écran. Vous pouvez opter pour :

`'arrow'` `'turtle'` `'circle'` `'square'` `'triangle'` `'classic'`.

• Par défaut, l'image est `'arrow'`. Pour changer : `shape('circle')`.

• La tortue est **visible** *(shown)* avec `st()` et **invisible** *(hidden)* avec `ht()`.

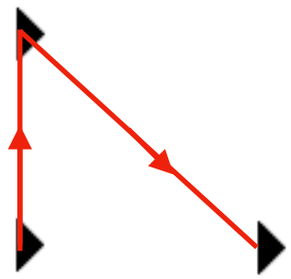
• Le **crayon** est baissé avec `down()` et levé avec `up()`. Sa couleur est changée avec `pencolor(c)` et la taille de sa mine avec `pensize(n)`.

• La fenêtre graphique est effacée et la tortue initialisée avec `reset()`.

1) Le graphisme cartésien (goto)

• Il utilise les **coordonnées cartésiennes** (x,y). La primitive de base est `goto(x,y)` ou bien `goto(M)` si M == (x,y). Pour dessiner une figure, il faut connaître les coordonnées des **sommets** de la figure !

```
clearscreen()
goto(0,100)
goto(100,0)
```

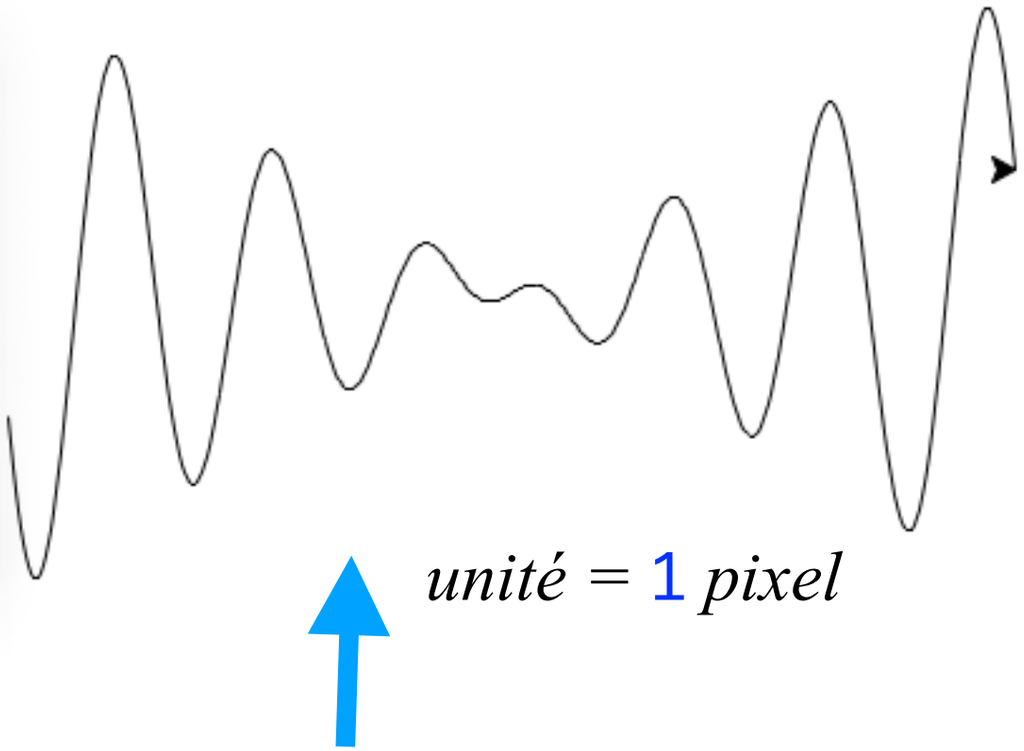


le cap est invariant par translation !

EXEMPLE : un mini **traceur de courbes** en cartésien.

• En programmation, une courbe est obtenue comme une suite de petits segments (un polygone). Nous dessinons ici la courbe d'une fonction $f : [a, b] \to \mathbb{R}$, avec un espacement h entre les abscisses sur lesquelles on calcule f. On prend h *petit*.
La courbe est en fait une succession de petits segments !

```
def dessiner(f,a,b,h):
    x = a ; y = f(x)
    up() ; goto(x,y) ; down()
    while x < b:
        x = x + h
        y = f(x)
        goto(x,y)     #  un petit segment
```




```
clearscreen()      #  effacement du canvas, tortue au centre
from math import cos
tracer(False)      #  pour ne pas aller à une allure de tortue…
dessiner(lambda x: 0.6 * x * cos(x/10), -200, 200, 1)
tracer(True)
```

2) Le graphisme **polaire** (forward, back, left, right)

• C'est souvent le plus intéressant. Une fois la tortue positionnée au point de départ du tracé, **on n'utilise plus les coordonnées cartésiennes**.

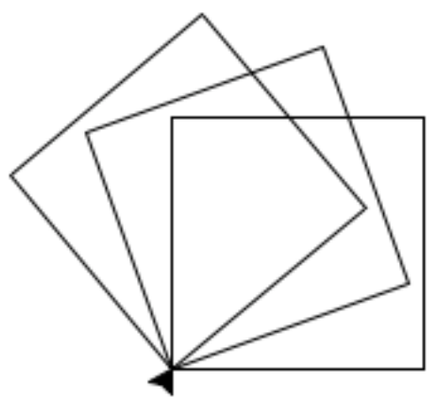
• La tortue se déplace dans le plan. Les deux **déplacements** élémentaires de la tortue sont :

- **avancer** ou **reculer** de d pixels avec l'instruction forward(d) ou back(d).
  Le cap ne change pas. Il s'agit d'une TRANSLATION. ***Modification de la position.***
- **tourner sur place** de a° (degrés !) avec l'instruction left(a) ou right(a).
  La position ne change pas. Il s'agit d'une ROTATION. ***Modification du cap.***

• Dessin d'un carré de côté c, là où est la tortue et dans la direction de son cap. Et trois fois, avec une rotation de 20° vers la gauche chaque fois.

```
def carre(c):
    for i in range(4):
        forward(c)
        left(90)
```

```
clearscreen()
for i in range(3) :
    carre(100)
    left(20)
```



Les primitives du graphisme tortue

POSITION	CAP	SE DEPLACER	CRAYON	FENETRE TORTUE
pos() goto(M)	heading() setheading(a°) towards(M)	forward(d) back(d) right(a°) left(a°)	down() up() pencolor(c) pensize(n)	clearscreen() bgcolor(c) color(c)

```
>>> pos()
(-25,50.35)
```

```
>>> right(90°)
>>> left(60°)
```

```
>>> A = (50,50)
>>> goto(A)
>>> goto(50,50)
```

```
>>> towards(A)
18.4349488229°
```

```
>>> forward(40)
>>> back(40)
```

```
>>> down()
>>> up()
```

```
>>> heading()
45°
>>> setheading(90°)
```

```
>>> dot(20)
```



```
>>> clearscreen()
>>> bgcolor('red')
>>> color('brown')
```

```
>>> bye()
>>> done()
```

```
>>> pencolor('blue')
>>> pensize(5)
```

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Importez toutes les fonctions du module `turtle`.
- Faites dessiner un triangle équilatéral de côté 50 puis un triangle rectangle de côtés 30, 40, 50.
- Programmez une fonction `polyreg(n,c)` dessinant à l'endroit où est la tortue et dans la direction de son cap courant un polygone régulier de n côtés, chaque côté ayant pour longueur c. Si n=4, ce sera un carré.
- Faites dessiner un cercle comme polygone à 36 côtés.
- Essayez de faire dessiner un bout de spirale.
- Programmez une fonction `marche(n)` faisant dessiner la marche aléatoire d'une tortue faisant des pas de longueur 2 et tournant à chaque pas d'un angle aléatoire entre -15° et +15°.
- Tirez des points A, B au hasard, dessinez le segment AB. Puis dessinez une portion de la médiatrice de AB.
- Illustrez avec la tortue la méthode de Monte Carlo vue en page 43. Pour afficher un point de diamètre d, utilisez l'instruction `dot(d)`.
- Le meilleur livre sur la tortue est ***Turtle Geometry***, paru au MIT Press, épuisé.

Les chaînes de caractères (str)

• Les **chaînes de caractères** sont utilisées pour présenter des résultats de calcul, mais il existe aussi des algorithmes propre aux textes (en français, ukrainien, chinois, braille, etc).

```
>>> s = 'Bonjour le Monde !'
>>> len(s)          # le nombre de caractères
18
>>> (s[0],s[1],s[2],s[-1])
('B', 'o', 'n', '!')
>>> 'jour' in s                          # test de sous-chaîne
True
>>> print(s[0],s[1],s[-1])               # sep=' ' par défaut
B o !
>>> print(s[0],s[1],s[-1],sep='')        # pas de séparation !
Bo!
```

```
>>> type(s)
<class 'str'>
```

• On peut **concaténer** (juxtaposer) des chaînes :

```
>>> 'Bon' + 'jour !'
'Bonjour !'
```

```
>>> 2 * 'Oui' == 'OuiOui'
True
```

Programmer sur des chaînes

• Le traitement du texte est un bon terrain de jeu pour s'entraîner aux **algorithmes non numériques** (messages secrets, langage naturel, IA, etc)…

```
def nbchiffres(s):   # nombre de chiffres d'une chaîne s
    res = 0
    for c in s:      # on boucle sur les caractères de s
        if c in '0123456789':
            res = res + 1
    return res
```

```
>>> nbchiffres('T23 = H628')
5
```

```
def poschiffre(s):                # position du premier chiffre de s
    for i in range(len(s)):       # on boucle cette fois sur les indices de s
        if s[i] in '0123456789':
            return i              # et on s'échappe !
    return False                  # sinon
```

```
>>> poschiffre('Les 2 ou 3 bateaux')
4
```

```
def nbchiffres(n) :  # nombre de chiffres d'un entier > 0
    return len(str(n))
```

```
>>> nbchiffres(78534200)
8
```

POUR ÊTRE SÛR D'AVOIR COMPRIS…

- Que pensez-vous être le résultat de `int('3')`, puis de `int('3+4')`, puis de `float('-2.783')`, puis de `float('67')` ? Vérifiez…

- Que pensez-vous être le résultat de `str(3)`, puis de `str(3+4)`, puis de `str(-2.783)`, puis de `str(67.0)` ? Vérifiez…

- Supposez que la fonction `len` (longueur) n'existe pas dans le type `str`. Comment pourriez-vous la programmer ? Nommez-la `lenbis`.

- Programmez une fonction `renverser(s)` renvoyant une copie de `s` mais à l'envers. Exemple : `renverser('Ch2K-36') == '63-K2hC'`.

- Programmez une fonction `str_int(s)` prenant une chaîne `s` ne contenant que des chiffres décimaux (base 10) et renvoyant l'entier qu'elle représente. Par exemple, `str_int('075360') == 75360`.

- Tous les caractères du monde sont codés avec UNICODE ! **Chaque caractère a un numéro unique** : `ord('A')` == 65, `ord('a')` == 97, `ord('!')` vaut 33, `ord('国')` est 22269 (*guó*). La fonction inverse de `ord` est `chr`. Construisez la chaîne `alphabet` `'abcdefghijklmnopqrstuvwxyz'` avec une boucle.