Ce mémento n'est pas un cours structuré ! Il va à l'essentiel.

Python en Seconde

(mémento)

Le programme Algo-Prog de **Seconde**

- Contenu de l'enseignement devant être acquis cette année :
- Le but visé est l'acquisition d'une **démarche algorithmique**.
  - La notion de **fonction.**
  - La **programmation** comme production d'un **texte** destiné à être **exécuté par une machine** (ordinateur, tablette, calculette…).

Le logiciel Python

www.python.org
www.thonny.org

Pydroid 3 (Androïd)
Carnets (iPad/iPhone)

TI-83 Python
(pour le BAC)

Qu'est-ce qu'un **algorithme** ?

• Ce n'est pas le langage Python qui est l'objet central de l'étude (mais il reste important !). Quel que soit le **langage** choisi pour **programmer**, il faut en maîtiser une (petite) partie pour **exécuter les algorithmes**…

• Un **algorithme** est une méthode de calcul mécanique destinée à résoudre une tâche donnée. Ce calcul sera effectué à la main ou par une machine, sans intelligence supplémentaire une fois l'algorithme trouvé.

• Mais pour trouver un algorithme, il faudra de l'intelligence (humaine) ! En tout cas, une certaine habitude de **résoudre des problèmes**. Et la pratique des maths est l'activité reine en la matière… Notre découverte des algorithmes sera donc assez largement tournée vers les maths.

• Dans la réalité, **les algorithmes sont présents partout** : banques, téléphones, Internet, musique, objets connectés (voitures, robots), etc.

• En fait, tous les algorithmes sont finalement basés sur les maths, sur les **nombres**. C'est l'aspect NUMÉRIQUE de la réalité.

Eléments du langage Python : les variables

- Le langage de programmation Python 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 la valeur d'une variable.

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

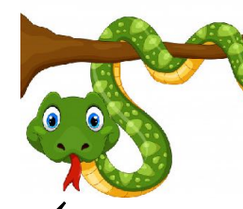
```
a ← 2
x1 ← 3 + a
```
langage naturel

```
>>> 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 ==

page 5

Eléments du langage Python : les **valeurs**

• La **valeur d'une variable** est un objet Python. Chaque ***objet*** appartient à une ***classe*** (ex: `int`) qui définit les *opérations* légales sur ses objets.

nombres		**booléens**	**chaînes de caractères**	**tuples**	**listes** → *(1ère)* ←
`-234`	`int`	`bool`	`str`	`tuple`	`list`
`2.34`	`float`	`True` `False`	`'Un élève !'`	`(3,2,'A')`	`[3,2,'A']`
`2+3j`	`complex`				

• 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**). Mais une variable peut changer de type au cours de l'exécution d'un programme !

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, trouve V=5 et à partir de maintenant x vaudra 5, jusqu'à sa prochaine affectation.

- **EGALITE**

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
```

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

Eléments du langage Python : les **fonctions** (1)

- Soit à définir en Python la fonction numérique $f : x \mapsto 2x-1$

- C'est une simple fonction mathématique réduite à une seule formule permettant de retourner un résultat. On peut se contenter de :

```
f = lambda x: 2*x - 1
```

la fonction x ↦ 2x-1

☜ x est le paramètre de f

```
>>> f
<function <lambda> at 0x1042f9e18>     # fonction <sans nom>…
>>> f(3)                                # 3 est l'argument de f
5
```

- Une **lambda-fonction** est une fonction **anonyme** (sans nom).

```
>>> (lambda x: 2*x - 1)(3) + 2
7
```

On peut l'utiliser sans lui donner de nom !

- Une lambda-fonction peut avoir plusieurs **paramètres** :

```
>>> g = lambda x,y: x - 2*y
```

```
>>> g(2,3)
-4
```

```
>>> g(2,0.5)
1.0
```

Eléments du langage Python : les **fonctions** (2)

- Soit à définir en Python la fonction numérique $(x,y) \overset{f}{\mapsto} 2x+y$.

- On utilise le mot-clé `def` de Python pour **rédiger** son texte, en général sur plusieurs lignes…

```
def f(x,y):
    return 2*x + y
```

☜ `x` et `y` sont les paramètres

☜ indentation (1 tabulation)

```
>>> f
<function f at 0x10139c1e0>
>>> f(3,4)
10
```

☜ cette fois-ci `f` a un nom !

☜ `3` et `4` sont les arguments

- C'est la manière usuelle de définir une fonction !

```
def g(x):
    return x - 2
```

```
>>> g(3)
1
>>> g(3.5)
1.5
```

```
>>> g(2,3)
TypeError
>>> g('oups')
TypeError
```

Eléments du langage Python : les **fonctions** (3)

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

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

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

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

```
>>> f(3,[4])
TypeError
```

• **Renforcer la sécurité** (léger) : On précise en commentaire les **types** des arguments attendus et celui du résultat. C'est de la responsabilité de l'utilisateur de la fonction de **respecter cette contrainte** !

un commentaire

```
def f(a,b):     # a,b sont des entiers naturels
    return 2*a + b
```

une docstring

```
def f(a,b):
    '''a et b doivent être des entiers naturels'''
    return 2*a + b
```

```
>>> help(f)
a et b doivent être des entiers naturels
```

Eléments du langage Python : les entiers (int)

• Chaque objet Python est d'un certain **type** (ou ***classe***). On dit que -13 est de type `int` pour signifier qu'il s'agit d'un nombre entier. *Ne croyez pas que* `int` *est l'ensemble mathématique (infini) de tous les entiers !...*

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

• Un type est muni d'**opérations légales sur les objets de ce type** : dans le type `int`, on trouve d'abord les opérations arithmétiques usuelles propres aux entiers, avec les **priorités** suivantes :

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

• Donc `2*3**9+5*4` est lu comme `(2*(3**9))+(5*4)`.

• Mettez des espaces pour la lisibilité : `2 * 3**9 + 5 * 4`

Eléments du langage Python : les entiers (int)

- IMPORTANT : la **division euclidienne** dans les entiers positifs.

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

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

"modulo"

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

a	b
r	q

43	5
3	8

- Essayez de n'utiliser ces opérations qu'avec des entiers > 0 pour ne pas vous compliquer la vie. Ne jamais se compliquer la vie 🤣...

- Un entier d **divise** un entier n si la division de n par d *tombe juste*, donc lorsque `n % d == 0`.

```
>>> 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`.

- Exemple d'**algorithme** : la **division** dans **N**.

Fonction **quotient** de a par b :

    res ← 0
    TANT QUE a ≥ b :
        res ← res + 1
        a ← a - b
    Résultat res

...en "langage naturel".

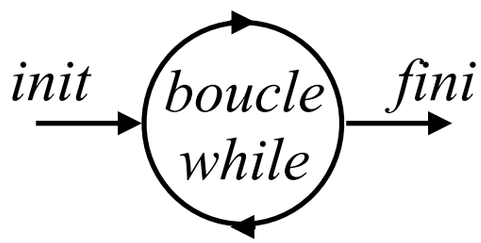
La flèche ← se lit "reçoit" ou "devient".

- Mais rien ne vaut un bon code <u>correct</u> et <u>exécutable</u> sur machine :

```
def quotient(a,b):
    '''a et b sont des entiers naturels, et b ≠ 0.
    Retourne le quotient de a par b.'''
    res = 0
    while a >= b:
        res = res + 1
        a = a - b
    return res
```

```
>>> quotient(15,3)
5
>>> quotient(17,3)
5
```

La boucle **while**



• En programmation, une "**boucle** while" permet de rédiger une **suite finie d'instructions** effectuant des **actions** (affichages à l'écran, modifications de variables, etc) et **se répétant** tant qu'une certaine **condition** reste vérifiée.

• On ne sait **pas** *a priori* combien de fois la boucle va tourner !

```
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
    d = 3                       # (sinon) init. du premier candidat
    while n % d != 0:           # TANT QUE d ne divise pas n:
        d = d + 2               #     d passe à l'impair suivant
    return d                    # on termine avec comme résultat d
```

• Une boucle travaille sur des **variables de boucle** (ici d) qui vont varier à chaque tour de boucle. Il faut s'assurer que la boucle TERMINE !

La conditionnelle if..elif..else..

• En rédigeant un algorithme, il faut constamment **prendre des décisions** suivant les données :

si x est positif, faire ceci, sinon faire cela…

```
if x > 0:
    faire_ceci
else:
    faire_cela
```

• En raisonnant avec des sous-cas, Python utilise la contraction `elif` pour "else: if" dont l'indentation à droite serait trop lourde :

```
Si x est positif :
    je fais ceci
    et ceci…
sinon :
    si x est nul :
        je fais cela
        et encore cela…
    sinon :
        je finis par ça…
```

BAD

```
if x > 0:
    ...
    ...
elif x == 0:
    ...
    ...
else:          # donc x < 0
    ...
```

GOOD

• En maths, exemple typique : une fonction en escalier…

La boucle bornée for

• En programmation, une "**boucle** for" permet de rédiger N fois une **suite finie d'instructions** effectuant des **actions** (affichages à l'écran, modifications de variables, etc).

• On **sait** *a priori* combien de fois la boucle va tourner !

```
def affcarrés(n):
    '''Affiche les carrés de 1² à n² avec n entier naturel'''
    for k in range(1,n+1):    # de 1 à n inclus
        print(k*k,end=' ')    # espace au lieu d'aller à la ligne
    print()                   # pour aller à la ligne à la fin !
```

```
>>> affcarrés(10)       # effet d'affichage mais aucun résultat !
1 4 9 16 25 36 49 64 81 100
```

• range(a,b) produit un par un les entiers de [a ; b-1].

• Une boucle travaille sur des **variables de boucle** (ici k) qui vont varier à chaque tour de boucle. La boucle bornée for TERMINE !

• Exemple d'**algorithme** : **nombre de diviseurs** d'un entier.

• Trouver le nombre de **diviseurs** d $\in [1; n-1]$ d'un entier $n \geq 2$.

• **Force brute** : j'essaye tous les entiers de 1 à n-1 inclus. Pour balayer un intervalle d'entiers, j'utilise une boucle for avec range :

```
def nbdiv(n):
    res = 0
    for d in range(1,n):
        if n % d == 0:
            res = res + 1
    return res
```

```
>>> nbdiv(12)
5
>>> nbdiv(13)
1
```

13 est un nombre premier !
(seulement divisible par 1)

range(1,n) produit les entiers de [1 ; n-1]

• **Force math** : Si $d < \sqrt{n}$ est un diviseur de n, il s'écrit n = d d' , donc d' est un autre diviseur distinct de d, qui vérifie $d' > \sqrt{n}$. Les diviseurs sont donc groupés par paires, sauf peut-être n/2. A vous de coder cette **accélération de programme** ! *Vous n'avez pas besoin de racine carrée…*

Pour n = 1000000, on fera 1000 tours de boucle au lieu de 1000000…

• Couples, triplets, etc. Le type **tuple** (1)

• Par chance, il est facile de former un **couple** en Python. De la même façon qu'en maths : `(a,b)`. Idem pour un **triplet** : `(a,b,c)`. Etc.

• Un couple c (ou un triplet, etc) forme en Python une **séquence** d'objets :

1. Les éléments sont **numérotés**. L'élément numéro `i` se note `c[i]` avec `i ≥ 0`.
   Les éléments d'un couple `c` sont donc `c[0]` et `c[1]`. ☜ c_0, c_1
2. On peut compter le **nombre d'éléments** avec `len(c)` . ☜ *length*
3. On peut savoir si `x` **appartient à** `c` avec l'opérateur `in` : `x in c`. ☜ $x \in c$
4. On peut **boucler** sur une séquence avec une boucle `for`.

• Quelques **séquences** d'objets sont étudiées en Seconde et Première. Elles sont caractérisées par les 4 points ci-dessus.

• Le résultat de `range(a,b)` est aussi une séquence. Les *chaînes de caractères* sont des séquences (p. 31). En Première, les *listes* seront de nouvelles séquences.

```
>>> r = range(8,15)
>>> (r[2], 13 in r)   # un couple
(10, True)
>>> len(r)
7                     # 15-8
>>> for x in r: print(x,end=' ')
8 9 10 11 12 13 14
```

• Couples, triplets, etc. Le type **tuple** (2)

• Les **tuples** de Python sont donc les couples, triplets, quadruplets, etc. des maths.

```
>>> c = (3,-2)
>>> type(c)
<class 'tuple'>
```

```
>>> len(c)
2
```

```
>>> c[0] + 2*c[1]
-1
```

• En Python, **une fonction n'a qu'un seul résultat**, ou **aucun**. Il peut être intéressant de retourner **plusieurs résultats**. Par exemple une division qui retourne à la fois le quotient et le reste. Comment faire ?

• Mais pardi, c'est bien sûr ! Un couple en résultat…

```
def division(a,b):
    '''a et b sont des entiers naturels, b ≠ 0.
       Retourne le couple (q,r)'''
    q = 0
    while a >= b:
        a = a - b
        q = q + 1
    return (q,a)
```

```
>>> division(15,3)
(5, 0)
>>> division(17,3)
(5, 2)
```

• Couples, triplets, etc. Le type **tuple** (3)

```
>>> d = division(17,3)
>>> d
(5, 2)
```

• Il reste à exploiter le résultat d qui est un **couple**. Pour cela, il y a deux possibilités :

- Soit extraire les deux **composantes numérotées** de d :

```
>>> d[0]
5
>>> d[1]
2
```

beurk...

- Soit utiliser la belle **affectation entre tuples** de Python :

de même longueur !

une déstructuration...

```
>>> (q,r) = d          # cool !
>>> q
5
>>> r
2
```

NB. Je PEUX faire cela car je SAIS que d est un couple !

Eléments du langage Python : les nombres approchés (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 = 0.0001, et contentez-vous de demander si |a - b| < h plutôt que a == b. Simple conseil d'ami…

• La valeur absolue `abs(x)` est standard, mais les autres fonctions de l'Analyse comme la racine carrée, le sinus, ou même le nombre π, etc. doivent être **importées** à partir du **module** `math` (p. 24).

• 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
```

F5

```
-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 on peut ne faire afficher que 2 ou 3 chiffres après le point décimal [PAA §3.4], ou utilisez* `round(x,2)`.

• Les variables globales

• Les variables définies en-dehors des textes de fonctions sont dites variables **globales**. La variable tva ci-dessous est globale :

```
tva = 18.6 / 100        # taux de TVA
```

• Python considère que les fonctions sont aussi des variables globales !

• Les variables globales sont **visibles** à l'intérieur d'un texte de fonction.

```
def prix_tcc(p):        # p est le prix HT
    return p * (1 + tva)
```

```
>>> prix_ttc(50)
59.3
```

• Pour **modifier** une variable **globale** à l'intérieur d'un texte de fonction, (en général une <u>mauvaise idée</u>), il faut utiliser le mot-clé **global** :

```
def modifier_tva(x):
    global tva
    tva = tva + x
```

```
>>> tva
0.18600000000000003
>>> modifier_tva(0.1)
>>> tva
0.28600000000000003
```

• Les variables locales d'une fonction

• Chaque **fonction** peut utiliser des **variables privées**, ce sont ses variables **locales**. C'est le cas pour `ma_tva` si celle-ci est déclarée à l'intérieur de la fonction `prix_restau_ttc` :

```
def prix_restau_ttc(p):    # p est le prix HT du menu
    ma_tva = 5.5 / 100     # variable locale
    return p * (1 + ma_tva)
```

• **Une variable locale n'est PAS visible à l'extérieur de la fonction** :

```
>>> prix_restau_ttc(25)
26.375
```

```
>>> ma_tva
'ma_tva' is not defined
```

privée !

• ATTENTION : une variable locale d'une fonction **masquera** une variable globale de même nom !

```
def prix_restau_ttc(p):
    tva = 5.5 / 100     # locale
    return p * (1 + tva)
```

```
>>> prix_restau_ttc(25)
26.375
>>> tva
0.28600000000000003
```

• Une **variable locale** dans 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 calculé une seule fois !
    return x2 / (1 + x2)
```

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

```
def g(x):
    x2 = x*x                    # x² calculé une seule 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, graphisme tortue, Internet,…), Python dispose de **modules** : bibliothèques de nouvelles fonctions et classes à **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 π, etc.

```
>>> from math import sqrt, pi
>>> sqrt(pi)
1.7724538509055159
```

$\sqrt{\pi}$

```
>>> import math
>>> math.sqrt(2)
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

• Le module random contient des fonctions permettant d'obtenir des nombres aléatoires (enfin, pseudo-aléatoires).

- 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 n faces** est randint(1,n).

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

F5 →

```
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]. Quasiment aucune chance de tomber sur a ou b…

```
from random import uniform
for i in range(2):
    print(uniform(0,1))
```

F5 →

```
0.7216978456973657
0.09891490615203236
```

Le module turtle

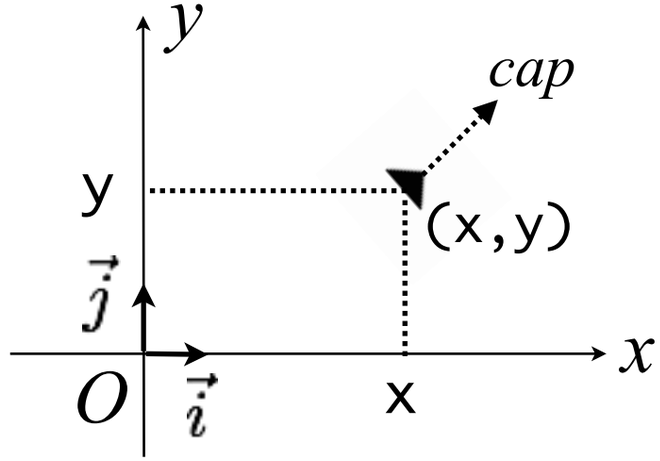
```
from turtle import *
```

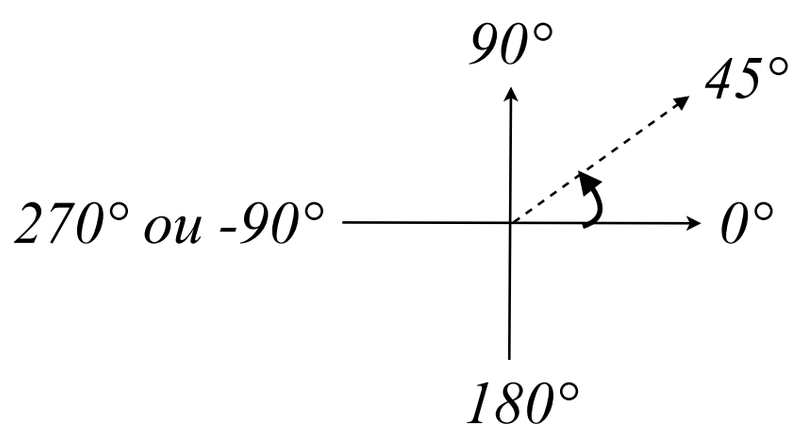
• Le programmeur pilote une tortue qui tient un **crayon** (*pen*). Elle est située à une **position** (x,y) dans un repère $(0, \vec{i}, \vec{j})$ orthonormé. L'origine 0 est au centre de la fenêtre graphique dont les dimensions sont par ex. 600 x 600 (pixels). Le sens positif des angles est celui des 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. Les réglages du départ ont lieu dans un fichier `turtle.cfg` se trouvant dans le même répertoire que vos programmes Python !

```
# turtle.cfg
width = 600
height = 600
shape = arrow
mode = standard
```

• 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()`.

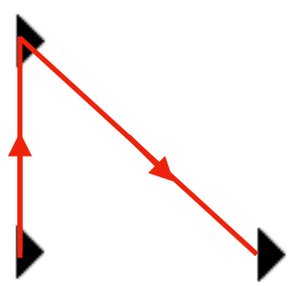
• Le **crayon** est baissé avec `down()` et levé avec `up()`. Sa couleur est changée avec `pencolor(c)` et la taille de sa mine avec `pensize(n)`.

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

1) Le graphisme **cartésien**

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

```
reset()
goto(0,100)
goto(100,0)
```

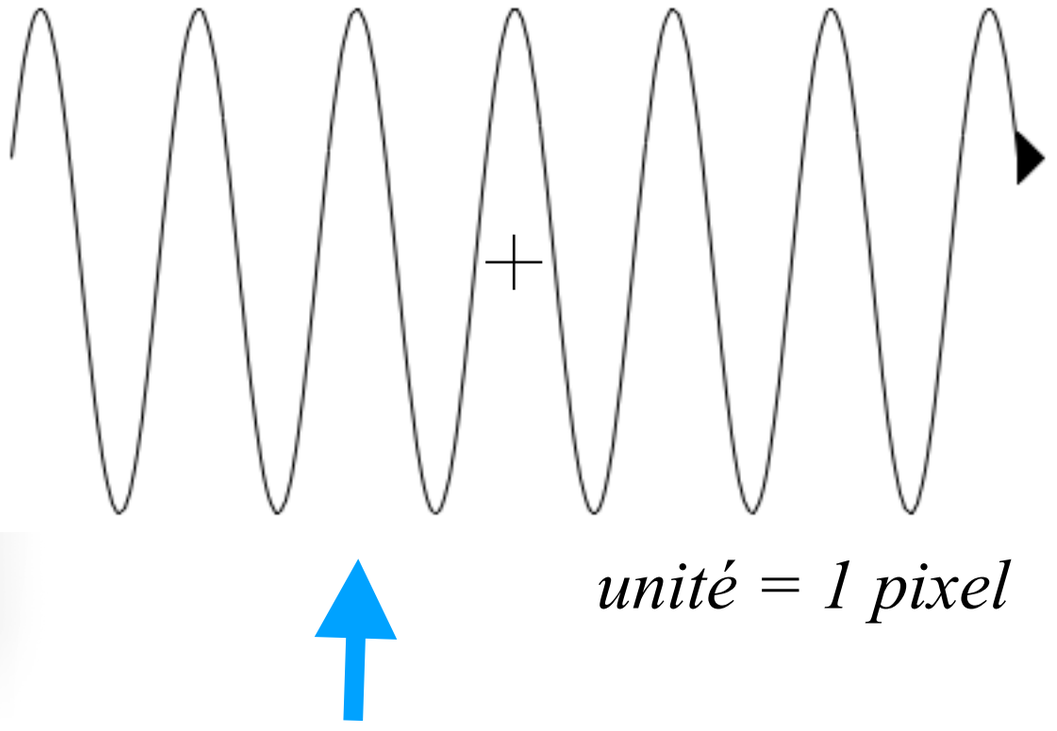


le cap est invariant !
(translation)

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".

```
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
```




```
reset()             # effacement du canvas, tortue au centre
from math import cos
tracer(False)       # ne pas aller à une allure de tortue…
dessiner(lambda x: 100 * cos(x/10),-200,200,1)
tracer(True)
```

amplitude *fréquence (écartement des spires)*

2) Le graphisme **polaire**

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

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

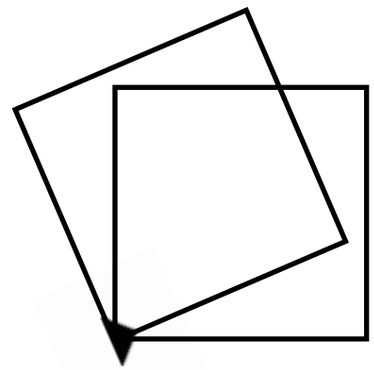
- **avancer** de d pixels avec l'instruction `forward(d)`. Le cap ne change pas. Il s'agit d'une TRANSLATION en direction du cap.
- **tourner sur place** de a° (degrés !) avec l'instruction `left(a)` ou `right(a)`. La position ne change pas. Il s'agit d'une ROTATION.

• Dessin d'un carré de côté c, là où est la tortue et dans la direction de son cap.

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

```
reset()
carre(100)
left(20)
carre(100)
```

F5



Les primitives 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)	reset() bgcolor(c) color(c)

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

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

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

```
>>> towards(A)
18.4349488229°
```

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

```
>>> right(90°)
>>> left(60°)
```

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

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

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

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, chinois, etc).

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

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

• On peut **concaténer** des chaînes :

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

```
>>> 2 * 'Bon'
'BonBon'
```

Programmer sur des chaînes

• Le traitement du texte est un bon terrain de jeu pour s'entraîner aux algorithmes non numériques.

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

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

```
def nbchiffres(s):    # une somme en compréhension, snob 😎
    return sum(1 for c in s if c in '0123456789')
```

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

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

SOMMAIRE