démarche didactique

1 Prérequis

Les élèves savent déjà programmer en Python, et en connaissent déjà tous les types et structures au programme.

2 Objectifs pédagogiques

- **Évaluer** un algorithme dans un contexte non calculatoire.
- **Abstraire** la syntaxe d'un langage pour établir une correspondance entre ce nouveau langage et python, et en appréhender les invariants et les différences. Au delà des aspects sémantiques, mettre le focus sur une différence de nature à classifier les langages en familles : ici, il s'agit de la gestion des types.
- Amener les élèves à constater qu'une saine maîtrise des types en entrée comme en sortie des fonctions est de nature à rendre l'algorithme plus sûr et plus lisible.
- *(plus secondaire)* Mettre en évidence qu'une sortie graphique permet souvent de rendre plus clair et plus explicite les actions de la machine, et à ce titre peut être utilisée pour débugguer un programme. On pourrait d'ailleurs l'exploiter par le truchement d'un module de type matplotlib.

3 Activité 1 - en mode débranchée

- Déroulé de l'activité : Lecture d'algorithmes, sur papier, dans un langage inconnu. Le travail peut être mené de façon individuelle ou en groupe, suivant l'aisance des élèves.
- Compétences travaillées : extrapoler le sens de cette nouvelle syntaxe pour pouvoir **évaluer** ces algorithmes qui produisent des dessins.
- Produire une synthèse via un tableau qui traduit de python vers asymptote (c'est le sens inverse des compétences mobilisées précédemment).
- Pour les élèves qui vont vite, avoir un dessin supplémentaire à disposition à traduire en langage Asymptote.

4 Transition vers l'activité 2

La différence la plus notable entre Python et Asymptote est la nécessité de typer toutes les données manipulées, ainsi que les fonctions. Nous devons amener les élèves à comprendre que

- le typage dynamique de Python est loin d'être majoritaire, il faut donc bien comprendre cette notion ;
- un intérêt plus pragmatique de typer et de rendre le code plus lisible et moins buggué ;
- typer les données permet de faciliter le travail de la machine dans la gestion de sa mémoire, et a donc un impact non négligeable sur les performances.

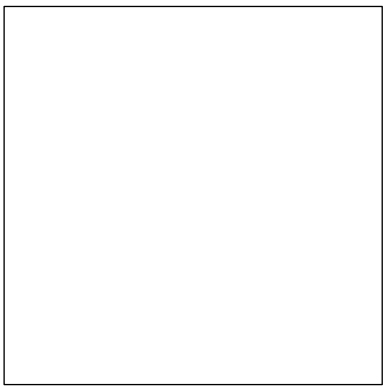
5 Activité 2 - sur machine (notebook jupyter)

- Déroulé de l'activité : les élèves sont sur un poste chacun, notebook ouvert. Les fragments de cours sont abordés et discutés avec l'enseignant. Ils sont entrecoupés d'exercices à aborder de façon individuelle.
- Compétences : on insiste sur les particularités de Python en regard des autres langages : La syntaxe (indentation et retour ligne) qui a déjà été comprise, mais on appuie sur l'importance du typage en programmation.
- On fait comprendre que cette notion est importante pour exécuter un programme avec sûreté et établir la validité des instructions au moment de la compilation.

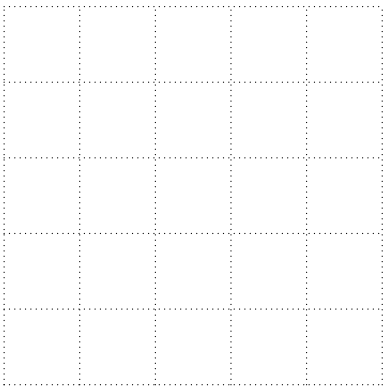
Un autre langage

Asymptote est un langage qui a pour but de produire des dessins. Voici différents scripts suivis par leur sortie. Mais les dessins n'ont pas été finis, car certaines lignes sont commentée par les caractères //. Évaluer ce code pour terminer les dessins.

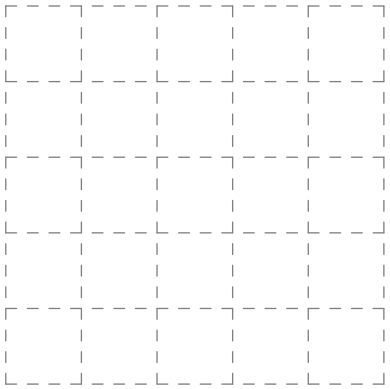
```
unitsize(5mm,5mm);
draw((10,0)--(0,0)--(0,10)--(10,10)--cycle);
for(int i=1;i<10;i=i+1){
  //draw((i,0)--(0,10-i),red);
  //draw((i,10)--(10,10-i),green);
}
```



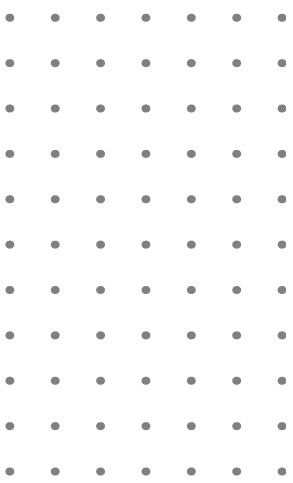
```
unitsize(10mm,10mm);
int[] ab={2,2,0,4,1,4,5,3,5,3};
int[] or={3,5,5,1,4,3,0,2,2,4};
string[] l={"R","A","T","N","U","A","G",
            "I","N","L"};
for(int i=0;i<6;i=i+1){
  draw((0,i)--(5,i),dotted);
  draw((i,0)--(i,5),dotted);
  }
for(int i=0;i<ab.length;i=i+1){
  //label(l[i],(ab[i],or[i]));
  //label(rotate(180)*l[i],(ab[i],5-or[i]));
  }
```



```
unitsize(10mm,10mm);
for(int i=0;i<6;i=i+1){
  draw((0,i)--(5,i),dashed+grey);
  draw((i,0)--(i,5),dashed+grey);
  }
pen f(int n){
  if(n % 6 == 0){return blue;}else{return yellow;}
}
for(int i=0;i<5;i=i+1){
  for(int j=0;j<5;j=j+1){
    path c=(i,j)--(i+1,j)--(i+1,j+1)--(i,j+1)--cycle;
    //fill(c,f(i*j));
  }
}
```



```
unitsize(6mm,6mm);
for(int i=0;i<7;i=i+1){
  for(int j=0;j<11;j=j+1){
    dot((i,j),grey);
  }
}
int i=0,j=0;
while(i+j<16){
  //dot((i,j),red);
  j=j+1;
  if(j>i+5){
    i=i+1;
    j=i;
  }
}
```



Compléter le tableau suivant :

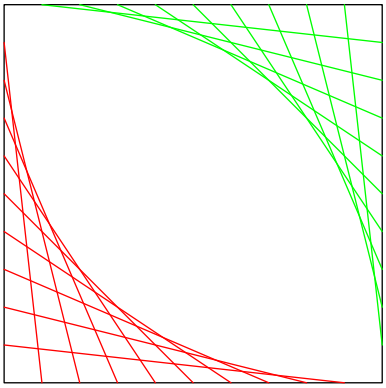
instructions en python	et l'équivalent en asymptote
`x=1.6` `y=4`	
`t=[5,2,8,7.5,3]`	
`for i in range(7):` `    ...........`	
`while x > 10:` `    ...........`	
`if x > 0:` `    .....` `else:` `    .....`	
`def f(x):` `    return([2*x,3*x])`	

Quelles autres différences peut-on remarquer ?

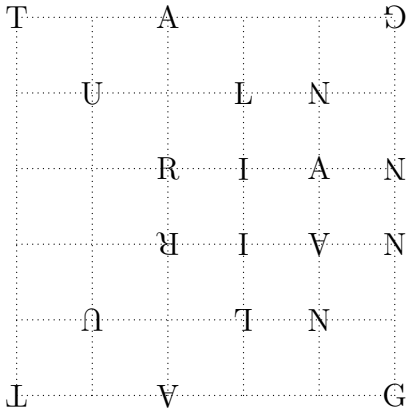
Un autre langage

Asymptote est un langage qui a pour but de produire des dessins. Voici différents scripts suivis par leur sortie. Mais les dessins n'ont pas été finis, car certaines lignes sont commentée par les caractères //. Évaluer ce code pour terminer les dessins.

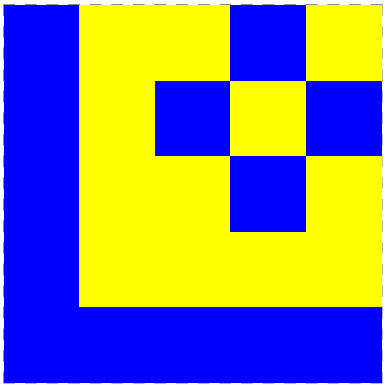
```
unitsize(5mm,5mm);
draw((10,0)--(0,0)--(0,10)--(10,10)--cycle);
for(int i=1;i<10;i=i+1){
  draw((i,0)--(0,10-i),red);
  draw((i,10)--(10,10-i),green);
}
```



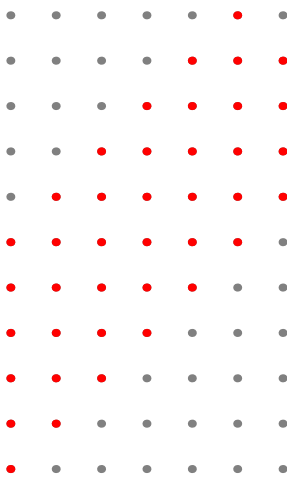
```
unitsize(10mm,10mm);
int[] ab={2,2,0,4,1,4,5,3,5,3};
int[] or={3,5,5,1,4,3,0,2,2,4};
string[] l={"R","A","T","N","U","A","G",
            "I","N","L"};
for(int i=0;i<6;i=i+1){
  draw((0,i)--(5,i),dotted);
  draw((i,0)--(i,5),dotted);
  }
for(int i=0;i<ab.length;i=i+1){
  label(l[i],(ab[i],or[i]));
  label(rotate(180)*l[i],(ab[i],5-or[i]));
  }
```



```
unitsize(10mm,10mm);
for(int i=0;i<6;i=i+1){
  draw((0,i)--(5,i),dashed+grey);
  draw((i,0)--(i,5),dashed+grey);
  }
pen f(int n){
  if(n % 6 == 0){return blue;}else{return yellow;}
}
for(int i=0;i<5;i=i+1){
  for(int j=0;j<5;j=j+1){
    path c=(i,j)--(i+1,j)--(i+1,j+1)--(i,j+1)--cycle;
    fill(c,f(i*j));
  }
}
```



```
unitsize(6mm,6mm);
for(int i=0;i<7;i=i+1){
  for(int j=0;j<11;j=j+1){
    dot((i,j),grey);
  }
}
int i=0,j=0;
while(i+j<16){
  dot((i,j),red);
  j=j+1;
  if(j>i+5){
    i=i+1;
    j=i;
  }
}
```



Compléter le tableau suivant :

instructions en python	et l'équivalent en asymptote
x=1.6 y=4	real x=1.6; int y=4;
t=[5,2,8,7.5,3]	real[] t={5,2,8,7.5,3};
for i in range(7):	for(i=0;i<7;i=i+1){ ..... }
while x > 10:	while(x>10){ ..... }
if x > 0: ..... else:	if(x>0){ ..... }else{ ..... }
def f(x): return([2*x,3*x])	real[] f(real x){ return({2*x,3*x}); }

Quelles autres différences peut-on remarquer ?

Notebook Jupyter

#
Différents langages
Il existe de très nombreux langages de programmation.
Ils se distinguent de différentes manières.

0.1 Niveau

Particularités	Niveau	Exemple
Plus simple et plus éloigné du fonctionnement de la machine	Haut	Python, Java
Plus complexe et plus proche du fonctionnement de la machine	Bas	C Assembleur Binaire

0.2 Paradigme

Parmi les langages de hauts niveaux, ils se distinguent de plusieurs manières - les objectifs sont différents (pour le calcul numérique, pour gérer et/ou interoger des bases de données ...) - les types des données manipulées sont différents

Liens - Schéma - rosettacode.org

1 Points communs :

La plupart de ces langages de programmation ont même "puissance" de calcul.

Pour Python cette propriété est assuré par les - déclaration de variable - affectation - séquence - tests - boucles

2 Particularités de Python :

Voici un programme écrit en C :

```
int factoriel(int n) {
    int resultat = 1;
    for (int loop = 1; loop <= n; loop++){
        resultat = resultat * loop;
    return resultat;
    }
}
```

Alors que, en Python, nous aurions :

```
In [ ]: def factoriel(n):
            resultat = 1
            for loop in range(1, n + 1):
```

```
            resultat = resultat * loop
        return resultat
```

```
In [ ]: factoriel(4)
```

2.1 Syntaxe

Comme vous avez pu l'observer, la syntaxe est fondamentale en Python : - Le **retour ligne** signale la fin d'une instruction - L'utilisation des deux points (:) et de l'**indentation** permet de délimiter les blocs

Python étant assez permissif, il est nécessaire d'être discipliné pour la mise en forme des programmes : les IDE nous aident beaucoup pour cela.

```
In [ ]: # code accepté mais pas du tout recommandé
        def factoriel(n):
            resultat = 1
            for loop in range(1, n + 1):
              resultat = resultat * loop
            return resultat

        factoriel(4)
```

```
In [ ]: # ceci n'est pas correct et est rejeté par Python
        def factoriel(n):
            resultat = 1
             for loop in range(1, n + 1):
                resultat = resultat * loop
            return resultat

        factoriel(4)
```

```
In [ ]: # Ce n'est probablement pas ce qui est attendu
        def factoriel(n):
            resultat = 1
            for loop in range(1, n + 1):
                resultat = resultat * loop
                return resultat

        factoriel(4)
```

2.2 Typage

Python est un langage dont le typage est **dynamique** : il n'est pas nécessaire de déclarer le type des variables.

Par ailleurs, il est capable d'inférer le type des objets créés : on parle de typage **fort**.

Tous servent à manipuler des données.

- Ce choix de typage peut être piegeux et "masque" la comprehension de l'execution machine : l'execution du programme est dépendante du type des variables et il est important de le prendre en compte (en Python, ce choix n'est pas, consciemment, fait par le concepteur du programme).
- Le contrôle à priori, de la validité d'un programme est moins évidente

Certains programmes peuvent donner des résultats inattendus (cela peut être non souhaité ou au contraire permettre de créer un programme plus général).

2.2.1 Paramètres

```
In [ ]: def mystere1(n):
            resultat = 0
            while n >= 0:
                resultat = resultat + n
                n = n - 1
            return resultat
```

Essayer d'affecter des valeurs de différents types (`int`, `float`, `bool`, `list`, `str`, `tuple`, ...) à la variable `var` correspondant au paramètre de la fonction `mystere1()`.

```
In [ ]: var = 5
        print('entrée :', var)
        print(type(var))
        retour = mystere1(var)
        print('sortie :', retour)
        print(type(retour))
```

Nous pouvons observer que la fonction accepte des paramètres de différents types.

- Etait-ce un choix?
- Est-ce un problème?

Exercice :

```
In [ ]: def mystere2(argument):
            resultat = argument[0]
            for elt in argument[1:]:
                resultat = resultat + elt
            return resultat
```

Testez la fonction `mystere2()` pour différents paramètres.

```
In [ ]: # A faire
```

Que fait cette fonction?
Exercice :

```
In [ ]: def mystere3(argument):
            resultat = argument[:0]
            for i in range(len(argument)):
                resultat = argument[i:i+1] + resultat
            return resultat
```

Reprenez la fonction `mystere3()` et testez la pour différents paramètres.

```
In [ ]: # A faire
```

Que fait cette fonction?

2.2.2 Retour de fonction

```
In [ ]: def tri(t):
            for i in range(len(t) - 1):
                for j in range(i + 1, len(t)):
                    if t[j] < t[i]:
                        t[i], t[j] = t[j], t[i]
```

```
In [ ]: T = [5, 8, 6, 9, 2, 3, 6, 4]
        liste = tri(T)
        print(liste)
```

Une fonction sert, généralement, à retourner une valeur.

Il faut retenir que la fonction retourne toujours une valeur. Par défaut, cette valeur est : `None`

Conclusion

Le typage dynamique et la **surcharge** de certains opérateurs peuvent fournir des résultats imprévus.

Python effectue certaines **conversions implicites** des types qui ne sont pas toujours naturels ou simples à comprendre.

Exemple :

```
In [ ]: print(1 + 1.2)
        print(True and '')
        print(True or '')
        print('' and True)
        print('' or True)
        print(True and 'Alan')
        print(True or 'Alan')
        print('Alan' and True)
        print('Alan' or True)
        print(True and [])
```

Cela n'est pas nécessairement un problème mais il est important de la prendre en compte : des résultats inattendus et non détectés peuvent se produire lors d'appels de fonction.

2.2.3 Spécification

Importance de la spécification et de la docstring (surtout en Python pour ce qui est du type)

```
In [ ]: def factoriel(n):
            """ n est un entier.
            Retourne l'entier n!"""
            resultat = 1
            for i in range(1, n + 1):
                resultat *= i
            return resultat
```

Exercice :

Reprenez la fonction `mystere2()` et donnez en la spécification.

2.2.4 Remarque : `assert`

Les `assert` peuvent aider à verifier le type.

```
In [ ]: def factoriel(n):
            """ n est un entier.
            Retourne n!"""
            assert type(n) is int, "Le paramètre est un entier"
            result = 1
            for i in range(1, n + 1):
                result *= i
            return result

In [ ]: # modifier le type de la variable
        factoriel(4)
```

2.2.5 Remarque : Type hints

Depuis la version 3.5, python supporte un mécanisme **optionnel** qui vous permet d'annoter les arguments des fonctions avec des informations de typage, ce mécanisme est connu sous le nom de *type hints*.

Voici un exemple d'utilisation :

```
In [ ]: def factoriel(n: int) -> int:
            """Retourne n!"""
            resultat = 1
            for i in range(1, n + 1):
                resultat *= i
            return resultat

In [ ]: factoriel(4)
```

Son intérêt est essentiellement de pouvoir documenter plus finement le code.

Certains IDE peuvent s'en servir pour permettre de vérifier le code et détecter des erreurs dans le passage d'arguments

Attention toutefois, cela reste optionel et est ignoré lors de l'interprétation du code.

```
In [ ]: def factoriel(n: str) -> str:
            resultat = 1
            for i in range(1, n + 1):
                resultat *= i
            return resultat

In [ ]: # Le typage n'est pas pris en compte à l'exécution
        factoriel(4)
```

On peut appeler cette fonction avec un `int` alors qu'elle est déclarée pour un `str`.

Pour aller plus loin

La notion de typage est fondamentale lorsque l'on élabore une fonction selon certaines spécifications.

Les **tests** à effectuer doivent, selon le langage utilisé, prendre en compte le type des variables et être en cohérence avec la façon dont les données sont représentées en machine.

Exercice

Il faut implémenter la fonction suivante, en fonction de sa spécification :

```
def racines(a, b, c):
    """a,b,c sont des entiers
    La fonction retourne les racines entières de ax^2 + bx + c = 0
    """
```

Solution

```
In [ ]: def racines(a, b, c):
            """a,b,c sont des entiers
            La fonction retourne les racines entières de ax^2 + bx + c = 0
            """
            solutions = set()
            if a == 0:
                sol = -c / b
                if c != 0 and int(sol) == sol:
                    solutions.add(int(sol))
                return solutions
            delta = b * b - 4 * a * c
            if delta == 0:
                sol = -b / (2 * a)
                if int(sol) == sol:
                    solutions.add(int(sol))
            if delta > 0:
                sol1 = (-b + delta ** 0.5) / (2 * a)
                if int(sol1) == sol1:
                    solutions.add(int(sol1))
                sol2 = (-b - delta ** 0.5) / (2 * a)
                if int(sol2) == sol2:
                    solutions.add(int(sol2))
            return solutions
```

On peut alors reflechir et mettre en place des tests, en considérant que les paramètres sont des entiers et les valeurs de retour des entiers.

```
In [ ]: def testRacines():
            # covers a>0, b<0, c>0, 2 small positive roots
            assert set(racines(1, -(5+8), 5*8)) == {8, 5}

            # covers b>0, c<0, 2 small negative roots
            assert set(racines(1, -(-8 + -2), -8*-2)) == {-2, -8}

            # a<0 , one root positive, other root negative
            assert set(racines(-7, -7*-1*(11+ -3), -7*11*-3)) == {11, -3}

            # covers c=0, one root zero, other root nonzero
            assert set(racines(1, -9, 0)) == {0, 9}

            # covers one nonzero root
            assert set(racines(1, -2*13, 13*13)) == {13}

            # covers b=0, both roots zero
            assert set(racines(1, 0, 0)) == {0}
```

```
        # covers roots as large as possible
        r1 = 10^3001
        r2 = -10^3001
        assert set(racines(1, -r1-r2, r1 * r2)) == {r1, r2}

    testRacines()
```

Sources : - Documents accompagnement NSI - Python 3 - des fondamentaux aux concepts avancés du langage (FUN) - Software Construction in Java (edX)
