COURS PI

L'école sur-mesure

de la Maternelle au Bac, Établissement d'enseignement privé à distance, déclaré auprès du Rectorat de Paris

Première - Module 3 - Programmation

Numérique et Sciences Informatiques

v.5.1

- ***Guide de méthodologie***
  pour appréhender notre pédagogie
- ***Leçons détaillées***
  pour apprendre les notions en jeu
- ***Exemples et illustrations***
  pour comprendre par soi-même
- ***Prolongement numérique***
  pour être acteur et aller + loin
- ***Exercices d'application***
  pour s'entraîner encore et encore
- ***Corrigés des exercices***
  pour vérifier ses acquis

www.cours-pi.com

Paris & Montpellier

EN ROUTE VERS LE BACCALAURÉAT

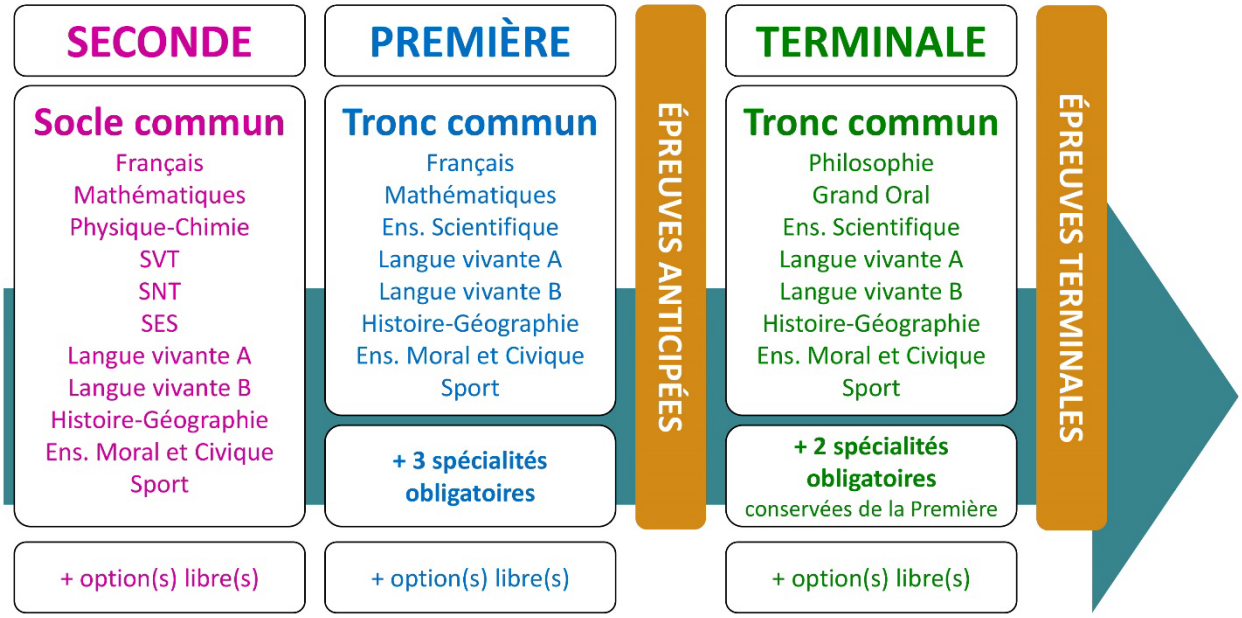
Comme vous le savez, **la réforme du Baccalauréat** est entrée en vigueur progressivement jusqu'à l'année 2021, date de délivrance des premiers diplômes de la nouvelle formule.

Dans le cadre de ce nouveau Baccalauréat, **notre Etablissement**, toujours attentif aux conséquences des réformes pour les élèves, s'est emparé de la question avec force **énergie** et **conviction** pendant plusieurs mois, animé par le souci constant de la réussite de nos lycéens dans leurs apprentissages d'une part, et par la **pérennité** de leur parcours d'autre part. Notre Etablissement a questionné la réforme, mobilisé l'ensemble de son atelier pédagogique, et déployé tout **son savoir-faire** afin de vous proposer un enseignement tourné continuellement vers l'**excellence**, ainsi qu'une scolarité tournée vers la **réussite**.

- Les **Cours Pi** s'engagent pour faire du parcours de chacun de ses élèves un **tremplin vers l'avenir**.
- Les **Cours Pi** s'engagent pour ne pas faire de ce nouveau Bac un diplôme au rabais.
- Les **Cours Pi** vous offrent **écoute** et **conseil** pour coconstruire une **scolarité sur-mesure**.

LE BAC DANS LES GRANDES LIGNES

Ce nouveau Lycée, c'est un enseignement à la carte organisé à partir d'un large tronc commun en classe de Seconde et évoluant vers un parcours des plus spécialisés année après année.



CE QUI A CHANGÉ

- Il n'y a plus de séries à proprement parler.
- Les élèves choisissent des spécialités : trois disciplines en classe de Première ; puis n'en conservent que deux en Terminale.
- Une nouvelle épreuve en fin de Terminale : le Grand Oral.
- Pour les lycéens en présentiel l'examen est un mix de contrôle continu et d'examen final laissant envisager un diplôme à plusieurs vitesses.
- Pour nos élèves, qui passeront les épreuves sur table, le Baccalauréat conserve sa valeur.

CE QUI N'A PAS CHANGÉ

- Le Bac reste un examen accessible aux candidats libres avec examen final.
- Le système actuel de mentions est maintenu.
- Les épreuves anticipées de français, écrit et oral, tout comme celle de spécialité abandonnée se dérouleront comme aujourd'hui en fin de Première.

A l'occasion de la réforme du Lycée, nos manuels ont été retravaillés dans notre atelier pédagogique pour un accompagnement optimal à la compréhension. Sur la base des programmes officiels, nous avons choisi de créer de nombreuses rubriques :

- **À vous de jouer** pour mettre en pratique le raisonnement vu dans le cours et s'accaparer les ressorts de l'analyse, de la logique, de l'argumentation, et de la justification
- **Pour aller plus loin** pour visionner des sites ou des documentaires ludiques de qualité
- Et enfin ... la rubrique **Les Clés du Bac by Cours Pi** qui vise à vous donner, et ce dès la seconde, toutes les cartes pour réussir votre examen : notions essentielles, méthodologie pas à pas, exercices types et fiches étape de résolution !

NUMÉRIQUE ET SCIENCES INFORMATIQUES PREMIÈRE

Module 3 – Programmation

L'AUTEUR

Adrien SAURAT

« L'enseignement, c'est favoriser l'autonomie et l'enrichissement des élèves, avec en autres objectifs, apprendre un métier. » Professeur et formateur en informatique avec plus de douze ans d'expérience en développement web et dans l'animation du réseau Canopé, il se passionne aussi pour le théâtre et l'écriture de nouvelles.
Des passions qui l'ont déjà conduit sur les planches du Festival d'Avignon.

PRÉSENTATION

Ce **cours** est divisé en chapitres, chacun comprenant :

- Le **cours**, conforme aux programmes de l'Education Nationale
- Des **applications** dont les **corrigés** se trouvent en **fin de chapitre**
- Des **exercices d'entraînement** et leurs **corrigés** en **fin de fascicule**
- Des **devoirs** soumis à correction (et ***se trouvant hors manuel***). Votre professeur vous renverra le corrigé-type de chaque devoir après correction de ce dernier.

Pour une manipulation plus facile, les corrigés-types des exercices d'application et d'entraînement sont regroupés en fin de manuel.

CONSEILS A L'ÉLÈVE

Vous disposez d'un support de cours complet : **prenez le temps** de bien le lire, de le comprendre mais surtout de l'**assimiler**. Vous disposez pour cela d'exemples donnés dans le cours et d'exercices types corrigés.
Vous pouvez rester un peu plus longtemps sur une unité mais travaillez régulièrement.

LES DEVOIRS

Les devoirs constituent le moyen d'évaluer l'acquisition de vos *savoirs* (« Ai-je assimilé les notions correspondantes ? ») et de vos *savoir-faire* (« Est-ce que je sais expliquer, justifier, conclure ? »).

Placés à des endroits clés des apprentissages, ils permettent la vérification de la bonne assimilation des enseignements.

Aux *Cours Pi*, vous serez accompagnés par un professeur selon chaque matière tout au long de votre année d'étude. Référez-vous à votre « Carnet de Route » pour l'identifier et découvrir son parcours.
Avant de vous lancer dans un devoir, assurez-vous d'avoir **bien compris les consignes**.
Si vous repérez des difficultés lors de sa réalisation, n'hésitez pas à le mettre de côté et à revenir sur les leçons posant problème. **Le devoir n'est pas un examen**, il a pour objectif de s'assurer que, même quelques jours ou semaines après son étude, une notion est toujours comprise.
Aux *Cours Pi*, chaque élève travaille à son rythme, parce que chaque élève est différent et que ce mode d'enseignement permet le « sur-mesure ».
Nous vous engageons à respecter le moment indiqué pour faire les devoirs. Vous les identifierez par le bandeau suivant :

Vous pouvez maintenant
faire et envoyer le devoir n°1

Il est **important de tenir compte des remarques, appréciations et conseils du professeur-correcteur**. Pour cela, il est **très important d'envoyer les devoirs au fur et à mesure** et non groupés. **C'est ainsi que vous progresserez !**

Donc, dès qu'un devoir est rédigé, envoyez-le aux *Cours Pi* par le biais que vous avez choisi :

1) Par **soumission en ligne** via votre espace personnel sur **PoulPi**, pour un envoi **gratuit**, **sécurisé** et plus **rapide**.
2) Par **voie postale** à *Cours Pi*, 9 rue Rebuffy, 34 000 Montpellier
   *Vous prendrez alors soin de joindre une **grande enveloppe libellée à vos nom et adresse**, et **affranchie au tarif en vigueur** pour qu'il vous soit retourné par votre professeur*

***N.B. :** quel que soit le mode d'envoi choisi, vous veillerez à **toujours joindre l'énoncé du devoir ;** plusieurs énoncés étant disponibles pour le même devoir.*
***N.B. :** si vous avez opté pour un envoi par voie postale et que vous avez à disposition un scanner, nous vous engageons à conserver une copie numérique du devoir envoyé. Les pertes de courrier par la Poste française sont très rares, mais sont toujours source de grand mécontentement pour l'élève voulant constater les fruits de son travail.*

SOUTIEN ET DISPONIBILITÉ

VOTRE RESPONSABLE PÉDAGOGIQUE

Professeur des écoles, professeur de français, professeur de maths, professeur de langues : notre Direction Pédagogique est constituée de spécialistes capables de dissiper toute incompréhension.

Au-delà de cet accompagnement ponctuel, notre Etablissement a positionné ses Responsables pédagogiques comme des « super profs » capables de co-construire avec vous une scolarité sur-mesure.
En somme, le Responsable pédagogique est votre premier point de contact identifié, à même de vous guider et de répondre à vos différents questionnements.

Votre Responsable pédagogique est la personne en charge du suivi de la scolarité des élèves.
Il est tout naturellement votre premier référent : une question, un doute, une incompréhension ? Votre Responsable pédagogique est là pour vous écouter et vous orienter. Autant que nécessaire et sans aucun surcoût.

QUAND PUIS-JE LE JOINDRE ?

Du lundi au vendredi : horaires disponibles sur votre carnet de route et sur PoulPi.

QUEL EST SON RÔLE ?

Orienter les parents et les élèves.
Proposer la mise en place d'un accompagnement individualisé de l'élève.
Faire évoluer les outils pédagogiques.
Encadrer et coordonner les différents professeurs.

VOS PROFESSEURS CORRECTEURS

Notre Etablissement a choisi de s'entourer de professeurs diplômés et expérimentés, parce qu'eux seuls ont une parfaite connaissance de ce qu'est un élève et parce qu'eux seuls maîtrisent les attendus de leur discipline. En lien direct avec votre Responsable pédagogique, ils prendront en compte les spécificités de l'élève dans leur correction. Volontairement bienveillants, leur correction sera néanmoins juste, pour mieux progresser.

QUAND PUIS-JE LE JOINDRE ?

Une question sur sa correction ?

- faites un mail ou téléphonez à votre correcteur et demandez-lui d'être recontacté en lui laissant un message avec votre nom, celui de votre enfant et votre numéro.
- autrement pour une réponse en temps réel, appelez votre Responsable pédagogique.

LE BUREAU DE LA SCOLARITÉ

Placé sous la direction d'Elena COZZANI, le Bureau de la Scolarité vous orientera et vous guidera dans vos démarches administratives. En connaissance parfaite du fonctionnement de l'Etablissement, ces référents administratifs sauront solutionner vos problématiques et, au besoin, vous rediriger vers le bon interlocuteur.

QUAND PUIS-JE LE JOINDRE ?

Du lundi au vendredi : horaires disponibles sur votre carnet de route et sur PoulPi.
04.67.34.03.00
scolarite@cours-pi.com

LE SOMMAIRE

OBJECTIFS

- Connaître des constructions élémentaires (affectations, conditions, boucles...).
- Savoir prototyper et coder une fonction avec ou sans arguments.
- Savoir utiliser des jeux de tests pour aider à la mise au point d'un programme.
- Avoir une idée du paysage des langages informatiques existants.
- Être capable de repérer des différences et points communs entre deux langages.
- Savoir utiliser la documentation d'un langage ou d'une bibliothèque de fonctions.

COMPÉTENCES VISÉES

- Programmer à l'aide de variables, de contrôles de flux et de boucles.
- Faciliter le regroupement de code à l'aide de fonctions paramétrées et sécurisées.
- Comprendre et utiliser des modules (bibliothèques) de fonctions complémentaires.
- Percevoir les différences et points communs pouvant exister entre différents langages.

CHAPITRE 2. Algorithmique 31

OBJECTIFS

- Différents algorithmes de parcours séquentiel.
- Comment effectuer une recherche dichotomique.
- Deux algorithmes de tri (par insertion, par sélection).
- La classification d'un élément grâce à l'algorithme des k plus proches voisins.
- La caractéristique d'un algorithme glouton, et un exemple d'usage.

COMPÉTENCES VISÉES

- Savoir coder le parcours séquentiel d'un tableau (algorithmes de recherche).
- Savoir coder un algorithme de tri (par insertion, par sélection).
- Expérimenter l'algorithme des k plus proches voisins.
- Savoir effectuer une recherche dichotomique dans un tableau trié.
- Résoudre un problème grâce à un algorithme glouton.
- Comprendre la notion de coût d'un algorithme.

SUGGESTIONS CULTURELLES

SITES RESSOURCES

- **www.numérama.com**
- **www.meta-media.fr**
- **www.pixees.fr**
- **Lumni** *: vidéos, articles, quizz*
  www.lumni.fr/lycee/premiere/voie-generale/nsi-numerique-et-sciences-informatiques

PODCASTS

- **Le code a changé** *France Inter*
- **La vie numérique** *France Culture*
- **Culture numérique**

ESSAIS

- **La pensée informatique** *Gérard Berry*
- **La cryptologie au coeur du numérique** *Jacques Stern*
- **Designing an Internet** *David D. Clark*

DOCUMENTAIRES AUDIOVISUELS

- **Une contre histoire de l'internet** *Sylvain Bergère*
- **Helvetica** *Gary Hustwit*
- **Citizenfour** *Laura Poitras*
- **Les ordinateurs du passé** *Le Vortex*


Le Cours

CHAPITRE 1
LANGAGES ET PROGRAMMATION

Depuis les langages de type assembleur apparus dès les années 50, il nous est possible de donner des ordres à un ordinateur en utilisant des commandes plus lisibles que le langage machine (cette suite de 0 et de 1 que nous ne saurions utiliser quotidiennement en tant qu'humains).

Il s'agissait de langages de bas niveau, en ce sens qu'ils étaient justement proches du fonctionnement de la machine (d'ailleurs un langage assembleur était spécifique à un processeur), mais nous disposons aujourd'hui de langages de plus haut niveau. Ces langages plus abstraits permettent d'utiliser le même code sur plusieurs machines différentes, sans problème (ou presque) de compatibilité.

Si l'utilisation de Python est préconisée au sein du cursus de lycée, c'est parce que ce langage est à la fois très répandu, facile d'accès et très modulable. En effet, malgré sa relative simplicité, il reste très utilisé dans le domaine de la recherche universitaire grâce au grand nombre de bibliothèques facultatives disponibles qui le rendent multi-fonctions.

Nous reviendrons donc sur la syntaxe de Python pour explorer les structures de base en programmation, et nous prendrons un peu de temps en suite pour comparer ce langage à d'autres de ses contemporains, afin de mieux cerner ce qui différencie un outil d'un autre.

OBJECTIFS

- Connaître des constructions élémentaires (affectations, conditions, boucles...).
- Savoir prototyper et coder une fonction avec ou sans arguments.
- Savoir utiliser des jeux de tests pour aider à la mise au point d'un programme.
- Avoir une idée du paysage des langages informatiques existants.
- Être capable de repérer des différences et points communs entre deux langages.
- Savoir utiliser la documentation d'un langage ou d'une bibliothèque de fonctions.

COMPÉTENCES VISÉES

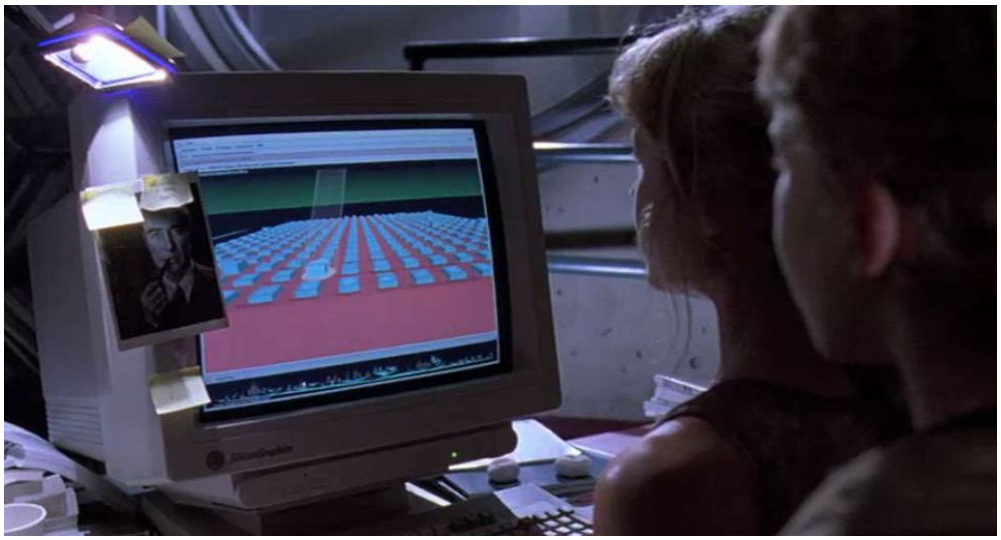
- Programmer à l'aide de variables, de contrôles de flux et de boucles.
- Faciliter le regroupement de code à l'aide de fonctions paramétrées et sécurisées.
- Comprendre et utiliser des modules (bibliothèques) de fonctions complémentaires.
- Percevoir les différences et points communs pouvant exister entre différents langages.

MATÉRIEL NÉCESSAIRE

- Un ordinateur permettant d'exécuter du code python.

Première approche
La programation au cinéma

Voici quelques images tirées de films mettant en scène des programmeurs et programmeuses !

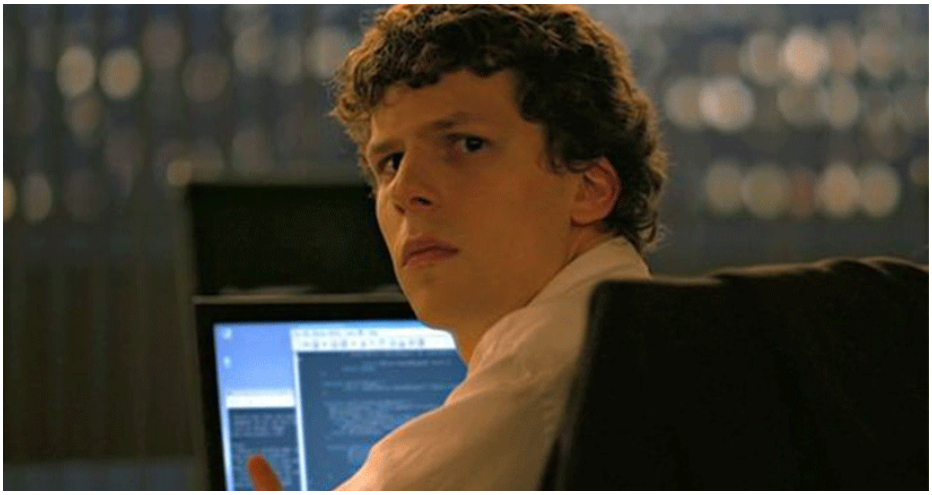


Document 1 : Jurassic Park, Steven Spielberg (1993)

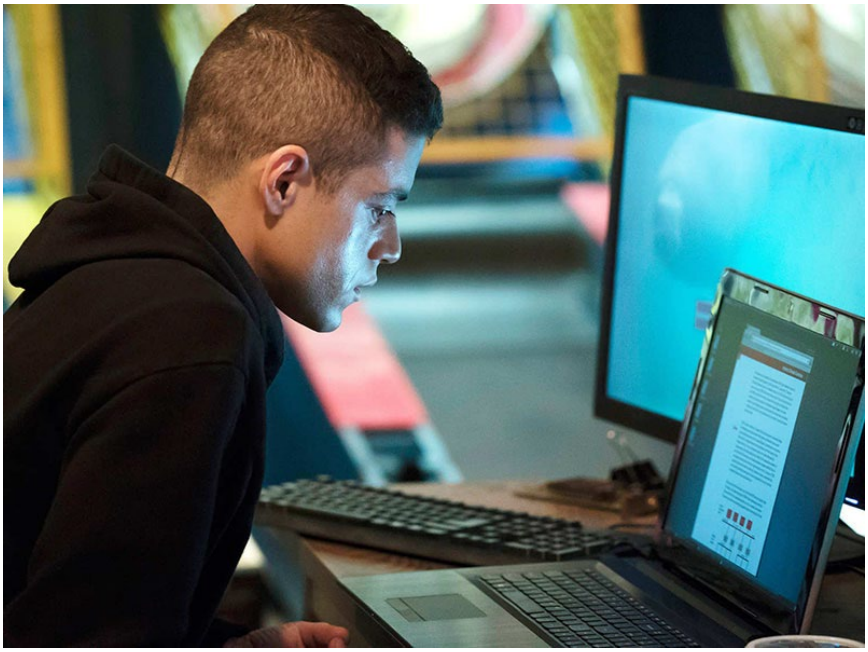
Document 2 : Matrix, Lana et Lili Wachowski

Document 3 : Office Space, Mike Judge (1999)

Document 4 : Wargames, John Badham (1983)



Document 5 : The Social Network, David Fincher (2010)



Document 6 : Mr. Robot, Sam Esmail (2015)

À VOUS DE JOUER 1

En vous basant sur ces quelques photos mais aussi et surtout sur les scènes que vous avez pu voir vous-même dans des films ou des séries, réfléchissez aux questions suivantes :

1. Comment est représentée la programmation dans les œuvres de fiction ?

2. Qui pratique la programmation dans ces œuvres ?

LANGAGES ET PROGRAMMATION
Variables et opérations courantes

Nous allons explorer quelques constructions élémentaires en programmation. Même si nous prendrons ici l'exemple de Python, gardez à l'esprit que de très nombreux langages utilisent les mêmes structures, avec simplement une syntaxe différente.

Certaines de ces notions sont déjà explorées depuis le programme de seconde (en SNT et en mathématiques) ou dans d'autres parties du programme de première. Ce chapitre mêle révisions, renforcement de bases et apprentissages de nouveaux concepts. Commençons justement en évoquant l'un des concepts les plus élémentaires et vitaux en programmation.

LES VARIABLES

Puisque les ordinateurs nous servent avant tout à traiter des informations afin d'accélérer toutes sortes de tâches, il nous est important de savoir comment stocker et manipuler ces informations.

Toute variable a un nom et une valeur. Cette valeur peut être de différents types : nombre, chaîne de caractère, tableaux de valeurs, etc. Le module 2 du programme de cette année explore plus en détail les différents types de données envisageables. Nous allons simplement revoir ici la façon dont on peut initialiser des variables simples.

On dit qu'on déclare une variable lorsque l'on procède à sa création, lui attribuant alors un nom et une valeur (dans certains langages on n'est pas obligé de choisir cette valeur initiale, mais elle prend alors une valeur par défaut, comme par exemple null, qui est un peu l'équivalent du zéro en arithmétique).

Voici quelques exemples de déclarations en Python :

```
varA = 1

x = 2.3

B = -4

variable_texte = "bonjour"

phrase = "Voici une variable de test"

info5 = True
```

Que pouvez-vous noter à propos des noms de variables ?

Et reconnaissez-vous les types de valeurs utilisés ici ?

NOMS DE VARIABLES

On peut constater que les valeurs ont des noms assez divers, contenant des lettres en majuscule ou en minuscule, parfois des chiffres ou même des signes spéciaux comme l'underscore. Aucun nom de variable ne contient d'espace.
Suivant le langage, les possibilités de nommage peuvent varier, mais nous sommes ici dans un cas assez standard où il y a peu de limitations tant que le nom de chaque variable tient en un seul bloc sans espace (ou sans autre signe de ponctuation comme la virgule, le point, etc.). N'hésitez pas à tester différents noms de variables sur votre ordinateur afin de voir ce qui est accepté par Python ou pas !

POUR ALLER PLUS LOIN

Des conventions de nommage facilitant le travail en équipe

Outre les limitations liées à ce que le langage utilisé peut accepter ou pas, il existe aussi des normes et règles que les programmeurs et programmeuses peuvent choisir de suivre afin de faciliter la lecture de leur code.

On peut ainsi noter trois grands types de conventions de nommage :

- Le **Camel Case**, avec un début de variable en minuscule, puis une majuscule au début de chaque mot supplémentaire (exemple : uneVariableTest) ;
- Le **Pascal Case**, identique au précédent si ce n'est que le premier mot contient aussi une majuscule (exemple : UneVariableTest) ;
- Le **Snake Case**, où les mots sont séparés par des underscores (exemple : une_variable_test).

Question : quelle convention de nommage vous paraît plus pertinente et à utiliser ?

Il n'y a pas de réponse universelle à cette question. Elle est spécifique à chaque personne, ou plutôt à chaque équipe, car bien sûr il est important, lorsque l'on travaille à plusieurs sur un même code, de partager les mêmes conventions de nommage.
Néanmoins, pour le cas de Python, notons sur le document PEP 8, alias « Style Guide for Python Code », préconise d'utiliser deux de ces normes ! Si l'on veut suivre cette recommandation, il convient d'utiliser le Snake Case pour les noms de variables et de fonctions, et le Pascal Case pour les noms de classe (une notion qui ne sera pas abordée en détail cette année). Pour l'essentiel de ce module (en dehors de quelques exemples spécifiques), nous suivrons cette recommandation.

EXERCICE 01

Admettons que nous souhaitions écrire un programme permettant de calculer les effectifs d'une école. Nous avons besoin dans une partie du code de stocker dans une variable le nombre d'élèves présents dans une classe. Comment nommeriez-vous cette variable ? (Utilisez plusieurs conventions de nommage)

TYPES DE DONNÉES

Si nous revenons à la liste de variables donnée plus haut en exemple, avez-vous reconnu les différents types déclarés ?

La variable varA contient un entier (valeur 1), x contient un nombre à virgule (valeur 2.3, avec un point comme séparateur, comme c'est l'usage en notation américaine), B un entier relatif (valeur -4). Nous avons ensuite deux variables contenant des chaînes de caractère, et une dernière qui contient un booléen (deux valeurs possibles dans ce cas : Vrai ou Faux, soit True ou False).

Remarquons au passage que toutes ces variables sont déclarées de la même façon. Ce n'est pas le cas dans tous les langages. Certains demandent à ce que le type de la donnée soit précisé lors de la déclaration. Python, quant à lui, détecte automatiquement le type de donnée. Mais il fait bien la différence entre un entier et une chaîne de caractère. Ici, on ne pourra pas faire les mêmes choses avec les variables varA (qui contient un entier) et variable_texte (qui contient une chaîne de caractères).

Dressons un tableau des types de base en Python :

Nom commun	**Type Python**	**Exemple**
Entier (positif ou négatif)	int	x = 10
Nombre flottant	float	x = 10.6
Chaîne de caractère (ou texte)	str	x = "Bonjour le monde"
Booléen	bool	x = True

Récapitulons aussi les types construits étudiés plus en détail dans le module 2 du programme de 1re :

Nom commun	**Type Python**	**Exemple**
Tuple ou p-uplet	tuple	x = ("kiwi", "fraise", "ananas")
Liste ou tableau	list	x = ["kiwi", "fraise", "ananas"]
Dictionnaire	dict	x = {"nom" : "Marc", "age" : 25}

Vérifier le type d'une variable

Vous pouvez utiliser la fonction **type** pour vérifier dans votre code le type d'une variable. Considérez par exemple le code suivant :

```
x = 5
print(type(x))

x = x * -1
print(type(x))

x = x * 1.5
print(type(x))
```

Avant de le taper, essayez d'abord d'imaginer quel serait le résultat de son exécution.

.....

.....

.....

.....

Puis tapez le et exécutez-le sur votre machine afin d'en constater les effets.

CONCATÉNATIONS ET CONVERSIONS DE TYPE

Maintenant que nous avons passé en revue les différents types de variable, voyons comment manipuler l'un d'entre eux : le type str, contenant les chaînes de caractères. Comment en combiner plusieurs, par exemple ? C'est quelque chose de très utile que l'on emploie couramment.

C'est une opération que l'on appelle concaténation. Dans chaque langage, l'opérateur permettant de faire ceci peut changer, en l'occurrence avec Python nous utiliserons le signe « plus ».

Prenons un premier exemple, que vous pouvez taper sur votre ordinateur :

```
a = "un"
b = "deux"
c = a + b
print(c)
```

Que constatez-vous ? Le résultat vous semble-t-il facile à lire ?

Souvent, on peut avoir intérêt à faire un mélange entre des variables et des morceaux de chaînes supplémentaires qui peuvent faciliter la lecture de l'ensemble. Exemple :

```
a = "un"
b = "deux"
c = a + " et " + b
print(c)
```

Qu'obtenez-vous cette fois-ci ?

Très bien. Maintenant, voyons comment peut fonctionner la concaténation entre une chaîne de caractères et un entier :

```
a = "un"
b = 2
c = a + b
print(c)
```

Quel est le résultat ?

Python ne semble pas beaucoup aimer cela, n'est-ce pas ? Nous avons pourtant vu plus haut que Python était capable de transformer un int en float lorsque c'était nécessaire. Oui, mais si c'est très naturel pour des traitements de chiffres, cela pose plus de soucis pour le langage lorsque nous traitons du texte. En fait, dans de nombreux cas, on peut avoir à indiquer à Python que nous désirons convertir une donnée d'un type en un autre type. Cette opération s'appelle **type casting** en anglais (conversion de type en français).

Reprenons donc l'exemple précédent et utilisons une fonction de conversion de type :

```
a = "un"
b = 2
c = a + str(b)
print(c)
```

Ici la fonction str transforme la variable qu'on lui donne en paramètre pour en faire une chaîne de caractères.

Que se passe-t-il lorsque vous exécutez ce code ?

Vous devriez voir apparaître la chaîne « un2 ».

Sachez qu'il est possible de convertir vers d'autres types :
- La fonction int() transforme une variable en entier ;
- La fonction float() transforme une variable en nombre flottant.

OPÉRATIONS ARITHMÉTIQUES

Nous avons vu que le signe « plus » permettait de concaténer des chaînes de caractère, mais bien sûr il est aussi utilisé en arithmétique. Voyons les opérations les plus courantes.

L'addition de deux entiers :

```
x = 1
y = 2
z = x + y
print(z)
```

EXERCICE 03

Addition d'un entier et d'une chaîne :

Même exemple, un peu modifié :

```
x = 1
y = "2"
z = x + y
print(z)
```

Avant de le taper, pouvez-vous en deviner le résultat ? Tapez-le ensuite pour vérifier.

Nous nous retrouvons dans un cas proche de ce qui a été vu plus haut pour les concaténations. Sauf qu'ici nous ne désirons pas obtenir une chaîne de caractère, mais obtenir un nombre : la somme entre 1 et 2. Comment faire cela sans modifier la déclaration de x et y ?

EXERCICE 04

Échange de valeurs

Supposons que nous disposons de deux variables déclarées ainsi : x = 5 y = 7

Comment faire pour arriver dans une situation où les valeurs sont inversées ? C'est-à-dire une situation dans laquelle x a pour valeur 7, et y a pour valeur 5. (Vous pouvez déclarer d'autres variables pour cela si nécessaire).

Outre l'addition, d'autres opérations sont bien sûr disponibles. Tapez-ceci et essayez d'en interpréter les résultats :

```
x = 9
y = 4

print("x + y = " + str(x + y))
print("x * y = " + str(x * y))
print("x - y = " + str(x - y))
print("x / y = " + str(x / y))
print("x % y = " + str(x % y))
```

Comprenez-vous toutes ces opérations ?

Les premières sont assez évidentes. Pour la dernière, il s'agit du « modulo », cette opération renvoie le reste de la division entière entre les deux variables utilisées pour le calcul.

Ici, 9 divisé par 4 sous une division entière renvoie un résultat de 2 (4 entre deux fois entièrement dans 9) et un reste de 1. C'est ce dernier reste qui est renvoyé par l'opérateur %. Y voyez-vous un usage ?

Cela peut permettre par exemple de vérifier qu'un nombre est pair. Si vous divisez une variable par 2 et que le reste de l'opération module est de 0, votre variable est paire. Si le résultat est égal à 1 (seul autre cas possible avec un diviseur de 2), la variable est impaire.

Note : nous avons vu ci-dessus les opérations les plus courantes, mais certaines plus spécifiques sont aussi prévues par Python. On peut ainsi déclarer un calcul de puissance avec le double astérisque ** (ex : 12**2 == 144) ou même réaliser une division entière avec // (ex : 6//4 == 1).

ASSIGNATIONS

Nous avons vu comment fonctionnaient les déclarations de variables en Python. Lorsque l'on donne une valeur à une variable, on dit qu'on lui assigne cette valeur. L'assignation de base a déjà été vue plusieurs fois, il s'agit par exemple d'un aaa = 1 qui assigne à la variable « aaa » la valeur 1.

Mais il existe aussi un moyen de mêler les opérations arithmétiques vues plus haut avec des assignations, afin de raccourcir un peu certaines commandes.

Si on veut par exemple faire passer la variable aaa à la valeur 2, nous pouvons écrire : aaa = aaa + 1
ce qui équivaut à lui attribuer la valeur 1 (qu'elle a au début de l'action) + 1, soit 2.

Mais on peut écrire cela de la manière suivante : aaa += 1
On peut faire la même avec chose avec les autres opérations.

EXERCICE 05

Assignations successives

```
x = 2
y = 5

x = x + 1
x += 2

y = y * 3
y *= 2

z = x + y
z -= 1

print(z)
```

Avant de le taper pour l'exécuter sur votre machine, essayez d'en suivre le déroulement de tête ou à l'aide d'un brouillon papier. Quel est le résultat affiché pour z à la fin ? (Vérifier ensuite en tapant le code pour l'exécuter vous-même).

02

LANGAGES ET PROGRAMMATION
Conditions et boucles

Nous allons explorer quelques constructions élémentaires en programmation. Même si nous prendrons ici l'exemple de Python, gardez à l'esprit que de très nombreux langages utilisent les mêmes structures et contrôle de flux.

CONDITIONS

Il est souvent utile de soumettre l'exécution de certains blocs de code à la vérification d'une condition précise. En pseudo-code, on peut décrire ce processus en français de la sorte :

- **Si** la condition1 est vraie
- **Alors** exécuter le blocA
- **Sinon** exécuter le blocB

L'équivalent Python de ces commandes ressemble à ceci :

```
# la variable condition1 doit avoir été
# déclarée auparavant

if (condition1 == True):
    print("Exécution du bloc A")
    # instructions A
else:
    print("Exécution du bloc B")
    # instructions B
```

Note : les parenthèses ne sont pas obligatoires ici après le if.

Que remarquez-vous au niveau de la vérification de la condition ? Le signe égal est doublé ! En effet, en Python et dans d'autres langages (pas tous), on différencie deux syntaxes :

- Le signe = seul permet d'affecter une valeur à une variable ;
- Le double signe == permet de vérifier l'égalité entre deux valeurs.

Il convient d'y veiller ! Essayez de coder un **if…else** en remplaçant le == par un =, vous constaterez que le résultat n'est pas correct, les deux cas possibles ne sont pas pris en compte !

Pile ou face !

On peut, dans Python, importer le module **random** qui permet d'utiliser certaines fonctions comme **randint**. Cette dernière pioche un entier au hasard (ce plutôt du pseudo-aléatoire, car il est difficile pour un ordinateur de faire mieux que ça).

La fonction randint accepte deux paramètres : en premier l'entier le plus bas que l'on peut obtenir, puis l'entier le plus haut.

Avec ces informations, écrivez un programme simple permettant de tirer à pile ou face. Lorsqu'on l'exécute, on a 50 % de probabilités d'obtenir le message « Pile », et 50 % de probabilités d'obtenir le message « Face ».

Nous avons examiné la façon dont les égalités sont constatées, mais bien sûr d'autres comparaisons sont possibles. Parmi les plus courantes on peut noter :

a == b	Vrai si a est égal à b
a > b	Vrai si a est strictement supérieur à b
a < b	Vrai si a est strictement inférieur à b
a >= b	Vrai si a est supérieur ou égal à b
a <= b	Vrai si a est inférieur ou égal à b
a != b	Vrai si a est inégal à b

Note : dans ces expressions, chaque partie (à gauche ou à droite) peut être une variable comme dans ces exemples, ou bien des calculs, des appels à des fonctions, etc.

Exemple :

```
if a + b > 5:
        #bloc d'instructions
```

C'est aussi dans ce genre de cas un peu plus complexes que le recours à des parenthèses peut devenir plus utile (afin de garder un code lisible) voire obligatoire pour que Python comprenne les relations logiques qu'entretiennent les expressions citées.

Ainsi, il est tout à fait valide d'écrire ceci :

```
a = False
b = True
c = False

if a == True and b == False:
    print("Ok")

if (a == True and b == False) or c == False:
    print("Ok2")

if a == True and (b == False or c == False):
    print("Ok3")
```

Si vous tapez ce code vous constaterez néanmoins que le placement des parenthèses a une influence sur le résultat ! Cela fonctionne un peu comme en mathématiques où 2 x (1 + 3) n'est pas égal à (2 x 1) + 3.

Comparaisons en série

Considérez le code suivant. Essayez de trouver, sans taper le code tout d'abord, quels seront les résultats affichés. Puis taper le code et comparez avec ce que vous avez trouvé. Vous pourrez ensuite modifier le programme afin de tester d'autres types de comparaisons (en intégrant notamment des chaînes et des conversions de type).

```
a = 1
b = 2
c = 3
d = 4
e = 2.0
f = 4.2

if (d == f):
    print("vrai")
else:
    print("faux")

if (b == e):
    print("vrai")
else:
    print("faux")

if (a > c):
    print("vrai")
else:
    print("faux")

if (d <= b):
    print("vrai")
else:
    print("faux")

if (d != a):
    print("vrai")
else:
    print("faux")
```

Boucles

En programmation, les boucles permettent de gagner du temps en répétant automatiquement certaines tâches, jusqu'à ce qu'une condition donnée soit remplie. On note deux grandes familles de boucles principalement : la boucle **for** et la boucle **while**.

Commençons par le cas **while** qui est peut-être le plus courant et le plus utile (tout dépend de ce que l'on veut faire, bien sûr). En anglais, ce terme signifie "tant que", et en effet ces boucles font la même chose tant qu'une condition indiquée n'est pas vérifiée. Attention donc aux boucles infinies ! Si la condition de sortie que vous écrivez n'arrive jamais, la boucle se contentera de faire travailler votre processeur sans terminer le programme. Il est alors nécessaire de couper son exécution de manière plus... autoritaire ! (Avec Ctrl+C dans la plupart des langages, ou l'accès au Gestionnaire des Tâches)

Prenons un exemple avec le code suivant. Tapez-le sur votre machine et exécutez-le :

```
compteur = 0

while compteur < 5:
    compteur = compteur + 1
    print("Le compteur a pour valeur " + str(compteur))
```

Que se passe-t-il si vous déplacez la ligne commençant par print pour la mettre au-dessus de celle qui fait augmenter le compteur ? Les valeurs affichées ne sont pas les mêmes ! C'est un exemple assez simple mais qui montre l'utilité du while.

Affichage d'un rang sur deux

Proposez un code (utilisant une boucle while) qui permettrait d'obtenir le résultat affiché suivant :

```
Le compteur a pour valeur 0
Le compteur a pour valeur 2
Le compteur a pour valeur 4
Le compteur a pour valeur 6
Le compteur a pour valeur 8
Le compteur a pour valeur 10
```

Vient ensuite la boucle **for**, qui pour sa part effectue des itérations sur une série de valeurs définie.

Commençons par un exemple simple utilisant une liste (un format de données étudié plus en détail dans le module 2 de cette année) :

```
print("Itération simple :")
fruits = ["ananas", "kiwi", "mangue", "poire"]
for i in fruits:
    print(i)
```

En tapant ce code, vous devriez voir les différents éléments de la liste être parcourus par la boucle for, ce qui est prouvé par le fait que tous font être affichés par le print. Modifiez maintenant le code pour qu'il ressemble à ceci :

```
print("Itération avec break :")
fruits = ["ananas", "kiwi", "mangue", "poire"]
for i in fruits:
    if i == "mangue":
        break
    print(i)
```

Comprenez-vous la différence ? La commande **break** qui a été ajoutée modifie le comportement de la boucle. Lorsqu'elle est exécutée, la boucle s'interrompt. Modifiez à nouveau le code pour obtenir ceci :

```
print("Itération avec continue :")
fruits = ["ananas", "kiwi", "mangue", "poire"]
for i in fruits:
    if i == "mangue":
        continue
    print(i)
```

Que constatez-vous ? Cette fois vous ne devriez pas voir l'itération s'arrêter, mais simplement omettre un élément. Lorsqu'on se trouve sur l'élément "mangue" la commande **continue** pousse le programme à aller à l'élément suivant, ignorant le print pour ce coup-ci.

En Python, il est possible d'effectuer une itération sur une simple chaîne de caractères (ce n'est pas le cas dans tous les langages). Cela peut donner par exemple ceci :

```
print("Itération sur une chaîne :")
for i in "ananas":
    print(i)
```

Bien souvent, c'est toutefois sur des plages numériques qu'on aura besoin d'effectuer une boucle. Même si les boucles while s'y prêtent bien, on peut tout à fait le faire aussi avec une boucle for, notamment grâce à l'utilisation de la commande **range**. Essayez ce code :

```
print("Itération sur une plage d'entiers :")
for i in range(5):
    print(i)
```

Notez que, si par défaut la commande range renvoie une séquence qui commence par zéro, il est tout à fait possible de commencer par un autre entier. En donnant deux arguments (deux paramètres entre parenthèses, séparés par une virgule), on définit donc le nombre de départ (inclu) et le nombre de fin (exclu). En fait, on peut même donner un troisième paramètre (qui par défaut a pour valeur 1) : la valeur d'incrémentation qui indique combien on doit ajouter au nombre précédent pour poursuivre la liste.

Exemples à reproduire sur votre machine (modifiez ensuite les paramètres pour expérimenter la fonction range) :

```
print("Itération sur des plages paramétrées.")

print("Deux paramètres : début et fin.")
for i in range(2, 6):
    print(i)

print("Deux paramètres : début, fin et incrément.")
for i in range(4, 12, 2):
    print(i)
```

Nous pouvons aussi utiliser les listes vues plus haut avec des nombres, et même y mélanger les types de valeurs.

```
print("Itération sur une liste de nombres :")
liste = [1, 2, 2.2, 4, 3, 10]
for i in liste:
    print(i)
```

EXERCICE 09

Réparation de boucle

Recopiez le code suivant :

```
liste = [1, 2, "3", 4.0, 5.1]
total = 0

for i in liste:
    i = i * 2
    print(i)
    total = total + int(i)

print("Total : " + total)
```

Vous devriez avoir du mal à l'exécuter tel quel, il contient quelques erreurs. Au moins une bloquante, et d'autres qui l'empêchent d'atteindre son but ! En fait, on voudrait obtenir dans le "Total" final la somme des nombres de la liste, tous multipliés arithmétiquement par deux.

Modifiez le code pour obtenir ce résultat ! Mais attention, vous n'avez pas le droit de modifier la déclaration de "liste". Laissez-donc la première ligne intacte. Pour le reste, vous pouvez ajouter, supprimer ou modifier des lignes.

LANGAGES ET PROGRAMMATION
Fonctions

Nous allons explorer quelques constructions élémentaires en programmation. Même si nous prendrons ici l'exemple de Python, gardez à l'esprit que de très nombreux langages utilisent les mêmes structures et contrôle de flux. En programmation, presque tous les langages permettent de déclarer des fonctions qui permettent de faire appel facilement à un morceau de code qui doit être utilisé fréquemment.

DÉFINITION ET APPEL D'UNE FONCTION

Le mot-clé **def** permet de définir une fonction, dont on peut choisir le nom. Prenons un exemple d'une fonction très simple qui se contente d'afficher un message fixe :

```
# Déclaration d'une fonction
def ma_fonction():
    print("Fonction personnalisée")

# Appel d'une fonction
ma_fonction()
```

ARGUMENTS

Bien souvent, l'appel à une fonction n'est pertinent que si on lui donne des paramètres pour personnaliser son action. Prenons un exemple très proche du précédent, mais qui ajoute un paramètre (qu'on appelle aussi argument) :

```
# Déclaration d'une fonction avec argument
def ma_fonction(param1):
    print("Affichage du paramètre : " + param1)

# Appel de la fonction avec paramètres différents
ma_fonction("Un")
ma_fonction("Deux")
ma_fonction("Trois")
```

ARGUMENTS MULTIPLES ET TESTS DE VALIDITÉ

Tout comme les fonctions proposées directement par Python, les fonctions personnalisées ne sont pas limitées à un seul argument ! Voyons un exemple qui en utilise deux :

```
# Déclaration d'une fonction avec arguments
def ma_somme(parametre1, parametre2):
    valeur_affichee = int(parametre1) + int(parametre2)
    print(valeur_affichee)

# Appel de la fonction de somme simple
ma_fonction(1, 2)
ma_fonction("10", 14)
ma_fonction("test", 11)
```

Une fois que vous avez tapé cette fonction, sans la modifier essayez de générer une erreur ou un résultat incorrect rien qu'en utilisant différents arguments (toujours au nombre de deux, mais de nature différente). Y parvenez-vous ?

Il s'agit d'un calculateur de somme (pas très utile puisque Python fait déjà ça très bien de base, ce n'est bien sûr qu'un exemple). Il transforme les paramètres en entier au cas où certains seraient envoyés en chaîne, mais il n'y a pas de gestion des floats, etc. Globalement il s'agit d'une fonction peu sécurisée qu'il est facile de rendre inopérante.

Avez-vous des idées pour l'améliorer ? Vous pouvez explorer cette fonction avant de passer à l'exemple suivant.

Pour ce qui suit, nous allons voir le cas d'une autre fonction de somme, qui contient quelques lignes supplémentaires dédiées à la vérification du bon format des paramètres en entrée. Elle ne fait pas de miracles non plus, mais permet au moins d'afficher un message d'erreur si on ne l'utilise pas correctement :

```
# Déclaration d'une fonction avec arguments
# Et contrôle de leur validité
def ma_somme(param1, param2):
    erreur = False
    if isinstance(param1, int) == False:
        erreur = True
    if isinstance(param2, int) == False:
        erreur = True

    if erreur == False:
        valeur_affichee = param1 + param2
        print("Somme : " + str(valeur_affichee))
    else:
        print("Erreur : paramètres non valides")

# Appel de la fonction de somme
ma_somme(1, 2)
ma_somme("10", 14)
ma_somme("test", 11)
```

Comme d'habitude, prenez le temps de recopier ce programme et de tâcher d'en comprendre toutes les lignes. Vous y constatez l'utilisation de la fonction **isinstance**, qui permet (un peu comme la fonction **type** que nous avons vu plus haut), de vérifier si une variable est d'un type donné.

Ici, donc, nous initialisons une variable "erreur" en début de fonction, avec la valeur False. Si un problème est constaté avec l'un des paramètres (ici, le fait que l'un d'entre eux n'est pas un entier), cette variable d'erreur passe à True.

Ceci permet de garder en mémoire que la somme n'est pas possible dans les conditions attendues (ici, on cherche à faire la somme entre deux entiers, et rien d'autre).

On peut donc effectuer une opération simple en fin de fonction :
- S'il n'y a pas eu d'erreur on effectue la somme et on en affiche le résultat ;
- S'il y a une erreur on affiche un message indiquant sa nature.

PORTÉE DES VARIABLES

Suivant où et comment une variable est déclarée, elle sera accessible dans certaines parties d'un programme et pas d'autres. Python n'est pas très contraignant à ce sujet, d'autres langages le sont plus.
Il est toutefois un cas relatif à ce qu'on vient de voir qu'il est important de garder en tête : une variable déclarée dans une fonction n'est utilisable que dans cette fonction.

Pour vous en rendre compte, recopiez le code suivant :

```
def fonction_test(param_entree):
    i = param_entree
    i += 1
    print(i)

fonction_test(1)
print(i)
```

Que se passe-t-il lorsque vous l'exécutez ? Si vous retirez ou commentez la dernière ligne, tout va bien, mais ce print(i) ne passe pas ! Et pour cause, la variable i n'est connu que dans la fonction, et pas en dehors, même si on a fait appel à cette fonction juste avant.

Notez qu'à l'inverse, une fonction ne peut pas utiliser d'autres variables que celles qu'elle reçoit en paramètres, et celles qui sont déclarées en son sein.

SPÉCIFICATIONS D'UNE FONCTION

Une spécification est pour une fonction ce qu'on pourrait qualifier de manuel. Elle permet de connaître le but d'une fonction, mais aussi la bonne façon de s'en servir : quels paramètres renseigner, avec quels types de variables, etc.

Pour prendre un exemple, voyons quelles sont les spécifications d'une fonction que nous avons utilisé précédemment. Tapez ceci dans la console Python :

```
>>> help(isinstance)
```

Vous devriez obtenir un retour de ce genre :

```
Help on built-in function isinstance in module builtins:

isinstance(...)
    isinstance(object, class-or-type-or-tuple) -> bool

    Return whether an object is an instance of a class or of a subclass thereof.
    With a type as second argument, return whether that is the object's type.
    The form using a tuple, isinstance(x, (A, B, ...)), is a shortcut for
    isinstance(x, A) or isinstance(x, B) or ... (etc.).
```

Les autres fonctions disposent de la même aide intégrée dans Python. Vous pouvez essayer avec une autre fonction que nous avons l'habitude d'utiliser :

```
help(print)
```

Ces textes d'aide sont contenus dans ce qu'on appelle la **docstring** d'une fonction. Vous pouvez déclarer une docstring dans les fonctions que vous créez, et c'est conseillé lorsqu'il s'agit de travailler sur un programme en collaboration avec d'autres personnes (dans le cas d'une participation à l'univers du logiciel libre, par exemple).

Il suffit pour cela de prévoir un texte contenu entre des guillemets successifs, de la manière suivante :

```python
def ma_fonction(param_chaine):
    """Cette fonction permet d'afficher
    un texte trois fois.

    Elle attend un paramètre obligatoire,
    de type str (chaîne de caractère)."""

    if isinstance(param_chaine, str):
        print(param_chaine * 3)
    else:
        print("Erreur : le paramètre doit être de type str")
```

Exécutez ce code puis tapez dans la console Python :

```
help(ma_fonction)
```

VALEUR PAR DEFAUT POUR UN ARGUMENT

Vous constaterez néanmoins que si l'appel à cette fonction avec un entier ou une chaîne fonctionne bien (dans le sens où l'entier renvoie un message d'erreur bien défini mais reste valide d'un point de vue Python), si vous appelez la fonction sans argument elle est en échec.

Nous avons vu plus qu'il était intéressant d'ajouter des tests de vérification de validité pour nos fonctions. Il est aussi possible de définir la valeur par défaut d'un paramètre.

Nous pouvons ainsi modifier cette même fonction de la sorte :

```python
def ma_fonction(param_chaine = ""):
    """Cette fonction permet d'afficher
    un texte trois fois.

    Elle attend un paramètre obligatoire,
    de type str (chaîne de caractère)."""

    if param_chaine == "":
        print("Erreur : paramètre obligatoire")
    elif isinstance(param_chaine, str):
        print(param_chaine * 3)
    else:
        print("Erreur : le paramètre doit être de type str")
```

La modification la plus importante se situe à la première ligne. C'est là qu'on décide que "param_chaine" sera au minimum une chaîne vide si cet argument n'est pas renseigné lors de l'appel à la fonction.

Les modifications suivantes prennent en compte cette possibilité pour renvoyer un message d'erreur personnalisé lorsque le cas se présente.

Notez l'utilisation de **elif** qui est une contraction de "else if" et qui permet de définir un cas du type "sinon si". On peut ainsi enchaîne plusieurs elif pour détailler de nombreux cas spécifiques d'un test.

ASSERTION

Pour aller encore plus loin en termes de recherche de robustesse lors de la création d'une fonction, examinons un autre outil à notre disposition : les **assertions**.

Il s'agit de conditions que l'on peut définir en début de fonction pour expliciter les conditions que doivent remplir les arguments pour être valides. Si, lors de l'appel à la fonction, ces conditions sont remplies, le code se déroule normalement. Si une condition au moins n'est pas remplie, le code est interrompu par Python qui renvoie alors une **AssertionError** qu'il complète avec le message personnalisé éventuel que l'on aurait joint à la commande **assert** (ce message est facultatif).

Exemple à reproduire et tester sur votre machine :

```
def ma_fonction(param1, param2):
    assert param1 > 0, "Paramètre 1 non conforme"
    assert param2 > 0, "Paramètre 2 non conforme"
    somme = param1 + param2
    print(somme)

ma_fonction(1, 0)
```

IMPLÉMENTATION DE JEUX DE TEST

Revenons un peu sur docstring pour aller un peu plus loin à ce sujet. En effet, il est possible d'y intégrer le résultat attendu de la fonction suivant différents paramètres d'entrée. Il sera alors possible ensuite d'utiliser un module spécifique, nommé **doctest**, afin de valider ces tests prédéfinis.

Prenons un exemple !

```
def fonction_operations(param1 = 0, param2 = 0):
    """Cette fonction attend deux nombres en paramètre,
    et effectue les opérations suivantes :
    - multiplication par deux du paramètre 1
    - ajout du paramètre 2
    - affichage du total

    Exemples :
    >>> fonction_operations(1)
    2
    >>> fonction_operations(1, 1)
    3"""
    total = param1 * 2
    total = total + param2
    print(str(total))


if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

La **première partie** est une déclaration de fonction, avec sa docstring. Notons tout de même que cette docstring contient des lignes d'exemple commençant par >>>. Ces lignes contiennent des commandes qui seront exécutées par le module de test ensuite. Chacune est suivie par une autre ligne qui indique le résultat attendu.

Ici en l'occurence, l'appel à fonction_operations avec un seul argument n'est pas tout à fait cohérent avec ce qui est décrit dans la docstring qui parle de deux paramètres. Mais cette ligne nous sera utile pour les tests, le temps d'un exemple. Bien sûr, dans le cas d'une fonction destinée à être utilisée de manière professionnelle, il est important de veiller à gérer un maximum de tests de validation (comme nous avons vu plus haut) et de s'assurer que la docstring colle parfaitement avec ce qui est attendu et proposé.

La **deuxième partie** est un appel au module doctest, suivi de son activation (avec l'option verbose qui lui demande de décrire en détail ce qu'il fait).

Dans ce cas précis, le résultat est le suivant :

```
Trying:
    fonction_operations(1)
Expecting:
    2
ok
Trying:
    fonction_operations(1, 1)
Expecting:
    3
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.fonction_operations
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

On peut voir que doctest a exécuté fonction_operations(1) et s'attendait à voir le résultat 2. C'est ce qu'il a obtenu en retour, validant ce premier test. Il a ensuite effectué le même appel de fonction avec les arguments (1, 1). Il attendait cette fois le résultat 3 et c'est là aussi ce qu'il a obtenu, validant le deuxième test.

Les deux tests prévus s'étant tous les deux déroulés correctement, il conclut avec le message "Test passed."

Mais à quoi ressemble un test échoué ? Modifiez votre code pour voir ça ! Il suffit par exemple de remplacer :

```
>>> fonction_operations(1)
    2
```

Par :

```
    >>> fonction_operations(1)
    4
```

Vous obtiendrez forcément une erreur !

Mais à quoi peut bien servir ce genre de test automatisé ?

Principalement à s'assurer qu'une fonction reste valide et cohérente, au fur et à mesure qu'on la fait évoluer. En effet, si vous modifiez votre fonction régulièrement pour en optimiser le fonctionnement ou pour y ajouter des fonctionnalités, ou bien encore pour y corriger des bugs, il n'en reste pas moins qu'elle devra toujours répondre d'une manière précise et pérenne à certains appels.

Ainsi, pour s'assurer qu'on n'a rien "cassé" après une mise à jour, l'appel aux tests de doctest permet de confirmer que le comportement primordial de la fonction est resté le même. Dans notre cas précis, si après avoir fait des modifications mineures ici et là on se rend compte que fonction_operations(1,1) ne renvoie plus 3, c'est qu'il y a un problème !

POUR ALLER PLUS LOIN

Test Driven Development

On appelle "développement piloté par les tests" le TDD qui consiste à mettre en avant le fait d'écrire d'abord un test pour chaque spécification du programme, puis d'écrire le code qui permettra de passer ce test avec succès.

Le TDD a été théorisé sous forme de trois lois :

- On doit écrire un test qui échoue avant de pouvoir écrire le code permettant de le faire réussir ;
- Il ne faut tester qu'un point précis du programme à la fois ;
- Il ne faut écrire qu'un minimum de code permettant de réussir le test.

Bien sûr, il faut un peu de temps et de pratique pour bien comprendre les subtilités de cette façon de programmer. Ce n'est pas la plus courante mais vous pourriez la rencontrer dans certains contextes professionnels.

Question : connaissez-vous d'autres méthodes de développement ?

Aucune méthode ne met tout le monde d'accord ! Chacune a ses avantages et ses inconvénients. On peut en citer d'autres comme le développement agile, l'extreme programming, etc. Certaines méthodes s'excluent naturellement, d'autres peuvent cohabiter. Il n'est pas important pour l'instant de les connaître, mais simplement de savoir qu'il existe différentes approches possibles, et que certaines sont plus adaptées que d'autres, suivant la taille et la nature de l'équipe, et/ou suivant le programme à réaliser.

04 LANGAGES ET PROGRAMMATION

Différents langages

DIFFÉRENCES ET POINTS COMMUNS

Chaque langage a été créé pour répondre à un besoin spécifique, et pour profiter des possibilités des machines de son temps. Leurs différences répondent aussi à de simples préférences de la part de leurs créateurs. Quoiqu'il en soit, les différences d'un langage à l'autre sont donc nombreuses, même si de nombreux concepts restent aussi communs à la plupart d'entre eux.

Mais rien ne vaut un exemple ! Nous allons comparer un même programme, écrit en Python puis en Perl, un autre langage qui a perdu en popularité mais reste utilisé à moins grande échelle. Voici la version Python :

```python
# Commentaire d'introduction
import random

aleatoire = random.randint(0, 3)
print("Résultat : " + str(aleatoire))

if aleatoire == 0:
    print("Zéro !")
elif aleatoire >= 2:
    print("Supérieur ou égal à deux.")
else:
    print("Un.")
```

Et voici donc l'équivalent en Perl :

```perl
#!/usr/bin/perl
use strict;

# Commentaire d'introduction
my $aleatoire = int(rand(4));
print "Résultat : " . $aleatoire . "\n";

if ($aleatoire == 0) {
    print "Zéro !";
}
elsif ($aleatoire >= 2) {
    print "Supérieur ou égal à deux.";
}
else {
    print "Un.";
}

```

EXERCICE 10

Comparer les syntaxes

Prenez le temps d'examiner ces morceaux de code. Notez les différences et les points communs que vous constatez.

Comme on peut le constater au travers de cet exercice, même s'il y a des différences notables, pour l'essentiel ce sont des choses faciles à adapter et qui ne changent pas radicalement la façon de programmer. Dans certains cas, les différences peuvent être plus profondes. Par exemple, si nous sommes ici dans une programmation procédurale, certaines font appel à la programmation objet (ou POO). Une autre philosophie, mais qui ne sera pas abordée dans ce cours.

05

LANGAGES ET PROGRAMMATION

Historique et environnement

Python propose de nombreuses bibliothèques de fonctions qui permettent d'en accroître les possibilités.

Il existe plusieurs moyens de savoir comment les utiliser. Déjà, sachez qu'une documentation officielle de Python existe en ligne, disponible gratuitement à l'adresse suivante :
https://docs.python.org/3/

Plusieurs langues y sont proposées, mais la traduction française n'est que très partiel et l'essentiel reste en anglais. Vous pouvez notamment aller y chercher la page intitulée "Global Module Index" qui liste tous les modules que l'on peut inclure dans notre code grâce à la fonction **import**.

Parmi ces modules, nous allons étudier l'exemple de la bibliothèque math, qui est souvent utile car elle propose des fonctions mathématiques courantes (on désigne par bibliothèque un ensemble de fonctions prédéfinies).

Avant tout, il faut faut importer le module ainsi :

```
import math
```

Vous pouvez le faire sous forme de programme à exécuter, ou directement dans la console.

Tapez ensuite dans la console :

```
dir(math)
```

Si vous voulez faire la même chose sous forme de script, il vous faudra plutôt taper :

```
print(dir(math))
```

En effet, **dir()** renvoie une chaîne mais ne l'affiche pas directement. Mais que contient ce résultat ? La liste des constantes et fonctions proposées par le module math.

Une constante, c'est un peu comme une variable, mais qui n'a pas vocation à évoluer. Elle contient une valeur fixe, c'est pour ça qu'on retrouve "pi" dans la liste, que vous pouvez avoir à demander, mais que vous n'irez pas écraser avec votre propre valeur !

Voici ce que le dir(math) contient donc :

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Pour faire appel à ces constantes et fonctions, il faut les prefixer par "math." et par exemple pour obtenir pi il faut donc utiliser **math.pi**. Essayez de l'afficher ! Evidemment, pi contient un nombre infini de chiffres après la virgule donc Python n'en propose qu'une approximation.

Mais si pour une constante l'utilisation est évidente, pour une fonction il peut en aller autrement. Prenons l'exemple de la fonction **floor**, comment savoir comment s'en servir ?

Eh bien, nous avons déjà vu comment obtenir des informations sur des fonctions de Python ! C'est avec **help** et nous allons encore nous en servir ici. La seule différence c'est que dans le cas d'une bibliothèque cela ne fonctionne que si nous l'importons avant et si nous pensons à préfixer correctement le nom de la fonction désirée.

Cela donne pour notre exemple (après avoir bien effectué l'import) :

```
help(math.floor)
```

Le résultat devrait être le suivant :

```
Help on built-in function floor in module math:

floor(...)
    floor(x)

    Return the floor of x as an int.
    This is the largest integral value <= x.
```

Cela décrit le fonctionnement de la fonction, elle attend un argument x et renvoie le plus petit entier inférieur à ce dernier. Essayez de faire des appels à floor du type...

```
math.floor(5.4)
```

Ou même...

```
math.floor(math.pi)
```

Très bien, mais c'est encore en anglais ! Si ça ne vous pose pas de problème, tant mieux. Vous aurez accès à toute la documentation incluse dans Python, ainsi que celle disponible sur le site officiel, et à d'innombrables forums et groupes d'aides disponibles sur internet.

Si ce n'est pas le cas, n'oubliez pas pour autant que Python est un langage très populaire qui dispose d'un grand nombre de ressources dans de nombreuses langues, y compris le français. L'usage d'un moteur de recherche avec des mots clés précis en relation avec votre questionnement du moment permet en général de trouver de quoi débloquer une situation.

Une carrière dans la programmation est forcément synonyme de veille technologique et de remise à niveau régulière. Avec en contrepartie l'assurance d'une routine jamais durable ! Dans tous les cas, si de nouveaux langages apparaissent très régulièrement, il n'en reste pas moins que d'autres langages (dont Python fait partie) restent pertinents et utiles pendant des années, voire des décennies.

Continuez à explorer les possibilités de ce langage, et profitez pour cela des nombreuses ressources qui lui sont consacrées et qui restent disponibles sur internet : cours complémentaires au format texte et vidéo, recueils de programmes à taper chez soi (et qui permettent d'explorer de nombreuses pistes de travail), forums de discussions, groupes Discord, etc.

L'avantage de Python sur le long terme, c'est qu'il est là pour durer et que son côté polyvalent lui permettra de vous être utile dans de nombreux métiers ou hobbies. Même si l'on ne devient pas programmeuse ou programmeur de profession, ce langage permet d'automatiser de nombreuses tâches répétitives, de programmer des jeux vidéo (avec Renpy, Pygame...), de réaliser des programmes embarqués (Raspberry Pi), etc. Les possibilités sont immenses !

Vous pouvez maintenant
faire et envoyer le devoir n°1