

Apprendre à
coder avec

Python



Sébastien Hoarau
Thierry Massart
Isabelle Poirier

Février 2020

Conditions d'utilisation du contenu du cours

CC-BY-SA : Sébastien Hoarau - Thierry Massart - Isabelle Poirier

Attribution - Partage dans les Mêmes Conditions

Les contenus peuvent être partagés et adaptés, y compris dans un but commercial, sous réserve de créditer l'oeuvre originale et de partager l'oeuvre modifiée dans les mêmes conditions.

Avant Propos	I
1 Bienvenue dans l'environnement Python 3	3
1.1 Première mise en contact	3
1.1.1 Sondage de début de cours	3
1.1.2 Bienvenue dans le cours en ligne : Apprendre à coder avec Python	3
1.1.3 Vos motivations et combien de temps allons-nous passer ensemble	4
1.2 Modalités pratiques du cours	7
1.2.1 Contenu de cette section	7
1.2.2 Informations générales sur le cours	7
1.2.3 Les modalités d'évaluation de notre MOOC	8
1.2.4 Naviguer dans notre MOOC	9
1.2.5 Aperçu de la suite de ce module	10
1.3 Mode d'emploi pour installer Python 3 et l'environnement PyCharm	10
1.3.1 Que va-t-on installer ?	10
1.3.2 Sur mon ordinateur Windows	12
1.3.3 Sur mon ordinateur MacOS	13
1.3.4 Sur mon ordinateur Ubuntu ou Linux	14
1.3.5 Si l'installation ne fonctionne pas	15
1.3.6 Installation d'un dictionnaire français dans PyCharm	16
1.3.7 Utilisation de PyCharm	16
1.4 UpyLaB, Python Tutor et la documentation officielle	22
1.4.1 Notre exercice UpyLaB	22
1.4.2 Exercice UpyLaB 1.1 - Parcours Vert, Bleu et Rouge	25
1.4.3 L'outil Python Tutor	27
1.4.4 La documentation officielle sur python.org	27
1.5 Au fait, c'est quoi un code ou un programme ?	28
1.5.1 Quelques définitions	28
1.5.2 Quiz de fin de module	29
1.6 Références et bilan du module	29
1.6.1 Références et bilan	29
2 Python comme machine à calculer et à écrire	31
2.1 Tout un programme au menu	31
2.1.1 Présentation du menu de ce module	31
2.2 Python comme machine à calculer	32
2.2.1 Valeurs et expressions	32
2.2.2 L'arithmétique	33
2.2.3 Tester l'arithmétique	34

2.3	Python comme machine de traitement de texte	35
2.3.1	Les expressions chaînes de caractères	35
2.3.2	À vous de tester les chaînes de caractères	36
2.4	Les variables pour changer	37
2.4.1	Les variables	37
2.4.2	Code avec variables	38
2.5	PyCharm en mode script, entrées et sorties	40
2.5.1	Manipuler des scripts avec PyCharm	40
2.5.2	Python Tutor et les diagrammes d'état	40
2.5.3	Commentons notre programme	41
2.5.4	Réalisation des exercices UpyLaB du module	44
2.5.5	Exercice UpyLaB 2.1 - Parcours Vert, Bleu et Rouge	45
2.5.6	Exercice UpyLaB 2.2 - Parcours Vert, Bleu et Rouge	46
2.5.7	Exercice UpyLaB 2.3 - Parcours Vert, Bleu et Rouge	47
2.6	Quelques fonctions prédéfinies, les modules math et turtle	48
2.6.1	Exemples d'utilisations de fonctions prédéfinies, et des modules math et turtle	48
2.6.2	Quelques fonctions prédéfinies et fonctions turtle très utilisées	51
2.6.3	Premier pavé pour le projet Vasarely	54
2.6.4	Exercice UpyLaB 2.4 - Non noté - Parcours Bleu et Rouge	56
2.7	Pour terminer ce module	58
2.7.1	Stockage des valeurs et caractères d'échappement	58
2.7.2	Les opérateurs d'assignation et de mise à jour	60
2.7.3	On passe à la pratique autonome !	61
2.7.4	Exercice UpyLaB 2.5 - Parcours Vert, Bleu et Rouge	61
2.7.5	Exercice UpyLaB 2.6 - Parcours Vert, Bleu et Rouge	62
2.7.6	Exercice UpyLaB 2.7 - Parcours Bleu et Rouge	62
2.8	Quiz de fin et bilan du module	63
2.8.1	Quiz de fin de module	63
3	Les instructions : tous vos désirs sont des ordres	65
3.1	Presque toutes les instructions au menu	65
3.1.1	Présentation du menu de ce module	65
3.2	L'instruction conditionnelle if : fais pas « si », fais pas ça !	65
3.2.1	L'instruction if	65
3.2.2	Les opérateurs relationnels	67
3.2.3	Exemples de code avec des if	68
3.2.4	Opérateurs logiques	69
3.2.5	Syntaxe et sémantique de l'instruction if	71
3.2.6	Quiz sur l'instruction if	74
3.3	Pratique de l'instruction if	74
3.3.1	Mettons ensemble en pratique ce que nous venons de voir	74
3.3.2	Jeu de devinette : proposition de solution	75
3.3.3	Le if : mise en pratique autonome : exercice UpyLaB 3.1	76
3.3.4	Exercice UpyLaB 3.2 (Parcours Vert, Bleu et Rouge)	78
3.3.5	Exercice UpyLaB 3.3 (Parcours Vert, Bleu et Rouge)	79
3.3.6	Exercice UpyLaB 3.4 (Parcours Vert, Bleu et Rouge)	80
3.3.7	Exercice UpyLaB 3.5 (Parcours Vert, Bleu et Rouge)	81
3.3.8	Exercice UpyLaB 3.6 (Parcours Vert, Bleu et Rouge)	82
3.3.9	Exercice UpyLaB 3.7 (Parcours Vert, Bleu et Rouge)	83
3.3.10	Exercice UpyLaB 3.8 (Parcours Rouge)	85
3.4	Les instructions répétitives while et for	87
3.4.1	L'instruction while	87
3.4.2	Syntaxe du while et calcul du plus grand commun diviseur	88
3.4.3	Conjecture de Syracuse	89
3.4.4	Deux canevas classiques rencontrés avec une boucle while	91
3.4.5	Des programmes qui bouclent indéfiniment	92

3.4.6	L'instruction for	92
3.4.7	Syntaxe du for, carrés, étoiles et autres polygones réguliers	94
3.4.8	Quiz sur while et for	96
3.5	Code avec while et for dans la pratique	96
3.5.1	La suite de Fibonacci	96
3.5.2	Un pavé du projet Vasarely avec des for	99
3.5.3	Mise en pratique autonome : exercices UpyLaB 3.9 et suivants	100
3.5.4	Exercice UpyLaB 3.10 (parcours vert, bleu et rouge)	101
3.5.5	Note sur la fonction print	102
3.5.6	Exercice UpyLaB 3.11 (Parcours Vert, Bleu et Rouge)	103
3.5.7	Exercice UpyLaB 3.12 (Parcours Bleu et Rouge)	104
3.5.8	Exercice UpyLaB 3.13 (Parcours Bleu et Rouge)	105
3.5.9	Exercice UpyLaB 3.14 (Parcours Rouge)	106
3.5.10	Exercice UpyLaB 3.15 (Parcours Rouge)	108
3.5.11	range(debut, fin, pas)	111
3.5.12	Exercice UpyLaB 3.16 (Parcours Rouge)	112
3.5.13	Exercice UpyLaB 3.17 (Parcours Rouge)	114
3.5.14	Exercice UpyLaB 3.18 (Parcours Rouge)	115
3.6	L'instruction pass et quiz de fin de module	116
3.6.1	L'instruction pass	116
3.6.2	Quiz de fin de module	116
3.7	Bilan du module	117
3.7.1	Qu'avons-nous vu dans ce module ?	117
4	Les fonctions : créons les outils que nous voulons	119
4.1	Au menu : comment fonctionne une fonction ?	119
4.1.1	Présentation du menu de ce module	119
4.2	Les fonctions prédéfinies et définies	119
4.2.1	Utilisation et définition de fonctions	119
4.2.2	Définition de nouvelles fonctions	121
4.2.3	Exécution de fonctions	122
4.2.4	Précisions sur les fonctions	123
4.2.5	Mini quiz	124
4.3	Faisons fonctionner les fonctions	124
4.3.1	Autres exemples de fonctions	124
4.3.2	Quelques aspects plus techniques	127
4.3.3	Première mise en pratique des fonctions	130
4.3.4	Mise en pratique des fonctions : exercices UpyLaB 4.1 et suivants	132
4.3.5	Exercice UpyLaB 4.2 (Parcours Vert, Bleu et Rouge)	133
4.3.6	Exercice UpyLaB 4.3 (Parcours Bleu et Rouge)	136
4.3.7	Exercice UpyLaB 4.4 (Parcours Bleu et Rouge)	138
4.4	Quelques règles de bonnes pratiques	139
4.4.1	Quiz "Testez vos connaissances !"	139
4.4.2	Code ou programme « propre »	139
4.4.3	Quelques éléments supplémentaires	142
4.4.4	Petit manuel des règles de bonnes pratiques	143
4.4.5	La règle du return unique et les instructions dont on ne veut pas prononcer le nom	144
4.5	Pratique des fonctions	144
4.5.1	Mettons ensemble en pratique ce que nous venons de voir	144
4.5.2	Autre exercice accompagné	146
4.5.3	Le yin et le yang revisité et paramètres par défaut	147
4.5.4	Un pavage du projet Vasarely avant déformation avec fonctions	149
4.5.5	Mise en pratique : exercice UpyLaB 4.5	151
4.5.6	Exercice UpyLaB 4.6 (Parcours Vert, Bleu et Rouge)	152
4.5.7	Exercice UpyLaB 4.7 (Parcours Vert, Bleu et Rouge)	153
4.5.8	Exercice UpyLaB 4.8 (Parcours Bleu et Rouge)	154

4.5.9	Exercice UpyLaB 4.9 (Parcours Bleu et Rouge)	156
4.5.10	Exercice UpyLaB 4.10 (Parcours Bleu et Rouge)	158
4.5.11	Exercice UpyLaB 4.11 (Parcours Bleu et Rouge)	159
4.6	Bilan du module	163
4.6.1	Qu'avons-nous vu dans ce module?	163

Index	163
--------------	------------

Bienvenue dans la session de printemps 2020 de notre MOOC intitulé [Apprendre à coder avec Python](#) diffusé sur la plateforme FUN.

Cette nouvelle version du cours est le fruit d'un travail important que nous avons réalisé depuis la première session diffusée en 2019.

Nous avons apporté les améliorations ou ajouts suivants :

- création d'un manuel comme support écrit pour accompagner le MOOC ;
- ajout de sous-titres aux vidéos (activables lors de la visualisation, mais attention, pas avec le navigateur Safari !);
- ajout d'une Foire Aux Questions (FAQ) pour vous permettre de trouver une réponse aux questions fréquemment posées ;
- ajout de résumés des vidéos du cours ;
- énoncés de notre exerciceur UpyLaB, plus clairs et structurés avec des exemples de ce qui est demandé et résultats des tests plus clairs ;
- choix des exercices UpyLaB mieux adapté au cours ;
- améliorations ou ajouts de certaines explications dans les textes ;
- horaire adapté : sur 15 semaines avec une ouverture de nouveaux modules toutes les 5 semaines (ouvertures des modules 1 à 4 à l'ouverture du cours, du module 5 en semaine 6 et du module 6 en semaine 11)

Nous vous invitons à vous [inscrire](#) dès à présent à cette nouvelle session pour bénéficier de l'ensemble des ressources.

Bon travail et bon amusement

Isabelle - Sébastien - Thierry

Février 2020

Bienvenue dans l'environnement Python 3

1.1 Première mise en contact

1.1.1 Sondage de début de cours

VOTRE AVIS NOUS INTÉRESSE

Note : Voir le sondage officiel FUN de la section 1.1.1

1.1.2 Bienvenue dans le cours en ligne : Apprendre à coder avec Python

QUI SOMMES-NOUS ET COMMENT ALLONS-NOUS VOUS APPRENDRE À CODER ?

Note : Voir la vidéo de la section 1.1.2 : Bienvenue dans le MOOC Apprendre à Coder avec Python

CONTENU DE LA VIDÉO

La vidéo précédente présente brièvement nos motivations pour vous apprendre le langage de programmation Python, l'approche proposée dans ce cours, les parcours et le projet proposés.

Tout ceci sera présenté plus en détails dans la suite de ce module de cours.

MODIFICATIONS POUR LA SESSION 3

Pour cette troisième session 3, l'horaire a été adapté : le cours s'étale maintenant sur 15 semaines avec une ouverture de nouveaux modules toutes les 5 semaines (ouvertures des modules 1 à 4 à l'ouverture du cours, du module 5 en semaine 6 et du module 6 en semaine 11); cela correspond à un travail hebdomadaire de 5 à 10 heures avec un projet évalué par les pairs. Si vous ne pouvez y consacrer 5h par semaine, vous pouvez également réaliser un parcours à votre allure, les modules restant accessibles même après la

fin du cours, mais vous ne pourrez bénéficier de l'évaluation du projet par les pairs ni de l'attestation finale délivrée par FUN après ces 15 semaines.

CRÉDIT : NOTRE OUTIL UPYLAB

Dans la suite du module nous présentons notre outil UpyLaB que vous utiliserez de façon intégrée dans ce MOOC pour réaliser de nombreux exercices de codage.

UpyLaB est le fruit de nombreuses années de développements d'informaticiens et étudiants de l'Université Libre de Bruxelles. Une bonne partie des développements et la coordination ont été réalisés de main de maître par notre informaticien, **Arthur Lesuisse**, que nous remercions vivement pour cela !

CRÉDIT : ÉQUIPES PEDAGOGIQUES ET MULTIMEDIA

Équipe CAP-ULB (anciennement « ULB-PODCAST » de l'ULB

Ce MOOC n'aurait pas vu le jour sans les membres de l'équipe ULB-Podcast qui nous ont accompagnés durant toutes les étapes, de sa conception à sa diffusion :

- Ariane Bachelart : accompagnement pédagogique et technique FUN
- Guillaume Gabriel : production multimédia
- Jérôme Di Egidio : production audiovisuelle
- Nathalie François : conseils pédagogiques
- Sophie Poukens : accompagnement pédagogique

Équipe de la Direction des Usages du Numérique de l'UR

Nous voulons également remercier les membres de l'équipe de la Direction des Usages du Numérique de l'Université de la Réunion (DUN) :

- Emmanuel Pons : production audiovisuelle
- Jean-François Février : production audiovisuelle

Bêta-testeurs et participants au teaser et à la première session

Merci aussi aux « Bêta-testeurs » de la première heure qui nous ont conseillés dans la réalisation de ce MOOC ainsi qu'aux personnes qui ont participé aux séances de confection du « teaser » : Ali Manzer, Anne-Françoise Biet, Arnaud Decostre, Ashley Yong Ching, Fabrice Nativel, Françoise Bols, Ludovic De Wachter, Marco Bianchin, Nicolas Pettiaux, Pierre Evrard, Priscillia Hardas, Ronico Billy, Vincenzo Sargenti, Yishen Yu.

Merci aussi aux participants des premières sessions de ce MOOC qui, par leurs commentaires, nous ont aidés à améliorer ce cours. Merci à tous !

1.1.3 Vos motivations et combien de temps allons-nous passer ensemble

NOS MOTIVATIONS

Nos premières motivations pour donner ce cours sont de vous montrer que programmer, c'est créer, et de vous transmettre le bonheur de cette activité.

CLÉ POUR RÉUSSIR CE COURS

Vous allez apprendre à programmer en utilisant le langage Python 3. Contrairement à d'autres matières comme l'apprentissage d'une langue étrangère ou des mathématiques, il s'agit probablement de quelque chose de réellement neuf pour vous.

L'apprentissage peut être comparé à l'apprentissage de la marche chez un enfant :

- L'enfant qui apprend à marcher va d'abord jouer dans son parc ou son bac à sable ; cela lui permet d'apprendre des gestes et de réaliser des choses utiles (ou non) pour cet apprentissage spécifique.
- Ensuite, quelqu'un va peut-être lui expliquer comment faire pour se lever, avancer, ...
- Plus tard, il va être aidé pour commencer à faire ses premiers pas.
- Et enfin, il va marcher et ensuite se lever seul et s'entraîner dur pour s'améliorer.

Nous vous invitons à faire le même type de parcours ici pour votre apprentissage du codage Python.

Pour réussir, il faut rigueur et persévérance et vous devez répondre oui aux questions suivantes :

- 1) Pourrez-vous consacrer suffisamment de temps, en respectant les échéances, pour cet apprentissage incluant visionnage des vidéos, lecture des explications, réponses aux quiz après mûres réflexions et codage des exercices demandés (entre 50 et 100 heures de travail en 15 semaines) ?
- 2) Êtes-vous prêt à apprendre une matière fort différente de ce que vous avez appris jusqu'à présent ?
- 3) Êtes-vous conscient que la persévérance est une clé fondamentale pour apprendre ?

MAIS QUELLES SONT VOS MOTIVATIONS POUR SUIVRE CE COURS ?

Et vous, pouvez-vous résumer vos motivations à suivre ce cours ? Écrivez dans les rectangles ci-dessous au maximum quatre mots ou phrases courtes (un par rectangle) qui représentent le mieux vos motivations et cliquez sur "enregistrer". Vous pourrez alors voir ce que les autres apprenants ont répondu.

Note : Le sondage est disponible en section 1.1.3 du cours en ligne

QUEL TEMPS POUVEZ-VOUS CONSACRER À CE COURS EN LIGNE ?

ET QUEL PARCOURS ALLEZ-VOUS SUIVRE ?

Ce cours en ligne, appelé aussi MOOC, est construit sur une base d'une dizaine de semaines de travail étalées sur 15 semaines. MOOC est l'acronyme de « Massive Open Online Course », prononcé « mouc », c'est-à-dire un cours ouvert, gratuit et massif en ligne.

Apprendre à programmer peut être fortement chronophage : le cours complet demande donc un rythme assez soutenu. Si vous ne pouvez ou ne voulez pas consacrer autant de temps chaque semaine à cet apprentissage, nous vous proposons un aménagement que vous pourrez éventuellement choisir de suivre n'importe quand tout au long du cours.

A priori le cours demande environ 12 heures de travail chaque semaine. C'est une évaluation moyenne. En réalité le temps que vous mettrez dépendra essentiellement de votre facilité à assimiler la matière et à réaliser les exercices, principalement de notre exerciceur UpyLaB.

De toute manière, si vous ne pouvez réaliser le travail complet, nous vous proposons deux parcours alternatifs.

Le principe est simple : tous les contenus théoriques, les exercices solutionnés et les quiz sont communs à tous les parcours. Par contre chaque exercice proposé par notre exerciceur UpyLaB a une ou plusieurs couleurs.

Un exercice dans :

- le parcours vert est un exercice de base (dans ce cas il sera aussi inclus dans les autres parcours) ;
- le parcours bleu est un exercice plus difficile (dans ce cas il sera aussi inclus dans le parcours rouge) ;
- le parcours rouge est un exercice jugé parmi les plus difficiles du cours, à faire si vous voulez suivre le parcours complet.

Libre à vous de moduler les exercices UpyLaB que vous réalisez, sachant que :

- Le parcours vert propose un travail moyen d'environ 6 heures par semaine.
Dans ce cas, a priori le projet final n'est pas inclus dans le cours.
Si vous suivez précisément le parcours vert, cela vous permettra d'avoir au maximum 61% des points. La réussite de la majorité des quiz et des exercices du parcours vert devrait donc vous permettre d'obtenir la note requise soit 50% des points ;
- Le parcours bleu propose un travail moyen d'environ 9 heures par semaine y compris un projet final ainsi qu'un travail d'"évaluation par les pairs" du projet de trois autres apprenants, et une auto-évaluation de votre projet.
Si vous suivez précisément le parcours bleu, cela vous permettra d'avoir au maximum 89% des points
- Le parcours rouge comprend l'ensemble des travaux et exercices, et propose un travail moyen d'environ 12 heures par semaine, y compris le projet final ainsi que l'évaluation du projet de trois autres apprenants et une auto-évaluation de votre propre projet.

Notez que la notion de parcours n'est ici qu'à titre informatif. Libre à vous de faire des exercices UpyLaB en dehors de votre parcours (par exemple des exercices du parcours bleu ou de réaliser le projet même si vous avez choisi le parcours vert).

Note : Veuillez à ne pas répondre aux quiz trop rapidement, puisque nombre d'entre eux ne permettent qu'un seul essai avant de noter votre réponse. Si votre réponse est fausse vous aurez alors définitivement perdu les points associés !

Enfin, si vous ne pouvez consacrer six heures par semaine, ou simplement si voulez suivre le cours et faire les exercices à un autre rythme, vous pouvez choisir de réaliser le parcours de votre choix à votre allure, sachant que vous pouvez obtenir une attestation FUN si vous avez obtenu 50% des points à la fin de ce cours, que le matériel et les exercices restent disponibles après la fin de la session, mais que les forums ne seront animés que pendant la période d'ouverture de la session et qu'il ne vous sera pas possible de soumettre le projet sur la plateforme ni d'obtenir l'attestation finale de succès après la clôture du MOOC.

CONSEILS POUR RÉUSSIR DANS VOTRE APPRENTISSAGE

- Soyez régulier dans votre apprentissage, par exemple en y consacrant une ou deux heures chaque jour.
- Si vous êtes bloqué dans la réalisation d'un exercice, n'hésitez pas à bien relire l'énoncé précis, à revoir encore et encore la matière déjà vue pour comprendre ce qui vous a échappé.
- La FAQ (Foire Aux Questions), organisée sous forme de questions d'apprenants et de réponses associées, est également une source d'information précieuse.
- Comme disait Nicolas Boileau en 1674 dans *l'Art Poétique* :

Vingt fois sur le métier remettez votre ouvrage,
Polissez-le sans cesse, et le repolissez,
Ajoutez quelquefois, et souvent effacez.

- Ne vous découragez pas et si rien ne va, adressez-vous aux forums pour chercher de l'aide.

COMBIEN D'HEURES PAR SEMAINE POURREZ-VOUS CONSACRER ?

- de 1 à 4 heures
- 5 à 7 heures
- 8 à 10 heures
- 11 heures ou plus

Choisissez l'option qui vous convient.

- Vous avez répondu « de 5 à 7 heures » : prenez le parcours vert
- Vous avez répondu « de 8 à 10 heures » : prenez le parcours bleu
- Vous avez répondu « 11 heures ou plus » : prenez le parcours rouge
- Vous avez répondu « de 2 à 4 heures » : prenez le parcours que vous désirez sachant que, si vous désirez une attestation de réussite FUN, vous devez obtenir 50% des points avant l'échéance du 22 mai 2020.

ET POURQUOI NE PAS NOUS EN DIRE UN PEU PLUS SUR LE FORUM ?

Nous allons passer du temps tous ensemble, pourquoi ne pas venir vous présenter dans le forum afin de faire connaissance ?

Venez nous raconter ce que vous faites dans la vie et pourquoi il est important pour vous de suivre ce cours.

Vous avez déjà utilisé le langage Python ou vous vous êtes déjà essayé à un langage informatique ? Partagez votre expérience avec la communauté.

ATTENTION !

Avant d'utiliser les forums, nous vous encourageons à prendre connaissance des règles et bonnes pratiques des forums via le lien <https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@BonnesPratiquesForum.pdf> et à les suivre scrupuleusement.

En particulier, ne mettez jamais d'informations personnelles comme votre nom, email ou n° de téléphone. Votre pseudo sera suffisant pour vous identifier.

Pour participer à la discussion :

1. **Cliquez sur « Afficher la discussion »** pour voir les messages déjà postés dans ce fil de discussion ;
2. Si un des fils traite du sujet dont vous voulez parler, **Cliquez sur « déplier la discussion »** en dessous du premier message posté ;
3. **Faites défiler la page** pour voir les autres messages postés et accéder à la boîte de rédaction ;
4. Éventuellement **cliquez sur « Nouveau message »** si vous désirez créer un nouveau fil de discussion sur un nouveau sujet, choisissez s'il s'agit d'une question ou d'une discussion et un titre pertinent,
5. Enfin **rééditez votre message** à l'endroit adéquat (nouveau message ou commentaire à un post existant ou dans la fenêtre étiquetée « Post a response ») et postez-le en cliquant sur « Soumettre » ou sur « Ajouter un message » pour ajouter un message au fil de discussion choisi ou nouvellement créé.

Note : Voir le forum *Faisons connaissance* dans la section 1.1.3 du cours en ligne

1.2 Modalités pratiques du cours

1.2.1 Contenu de cette section

CONTENU DE CETTE SECTION

Dans cette section, nous donnons des informations générales sur notre MOOC (objectifs, calendrier, modalités d'évaluation), sur la manière d'utiliser la plateforme d'apprentissage FUN ainsi que les modalités de communication entre apprenants et l'équipe.

La section est terminée par une vidéo qui fait une rapide présentation de la suite de ce présent module.

1.2.2 Informations générales sur le cours

FICHE DU COURS ET CALENDRIER

Cette section reprend ou donne accès aux informations générales du cours y compris l'horaire de diffusion des différents modules et les échéances principales.

Fiche du cours

Elle donne la plupart des informations générales sur le cours, et est accessible via le lien que vous pouvez trouver ici (<https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session03/about>)

Attestation de suivi avec succès

Tout apprenant atteignant un taux de réussite de 50% des points recevra une attestation de suivi avec succès produite et envoyée par la plateforme FUN.

Calendrier et échéances principales

Le calendrier et échéancier du MOOC est donné dans l'onglet Calendrier.

Vous pouvez également visualiser le calendrier en cliquant sur le lien ici (<https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session03/pdfbook/1/>).

En particulier, vous devez impérativement respecter les trois échéances suivantes pour valider votre apprentissage :

- réalisation des quiz et exercices *UpyLaB* des modules 1 à 6 avant l'échéance finale.

Si vous décidez de faire le projet :

- remise du projet proposé en module 5 avant le 20/04/2020 ;
- évaluation de projets de pairs avant le 11/05/2020 ;
- auto-évaluation de votre projet amendé avant l'échéance finale du 22/05/2020.

Notez que les procédures de remise du projet et d'évaluations sont dans le module « Projet ».

Le module projet correspond au module 7 de ce support de cours.

1.2.3 Les modalités d'évaluation de notre MOOC

PARCOURS

Comme expliqué précédemment, trois parcours vous sont proposés, le parcours vert, le parcours bleu ou le parcours rouge. D'un autre côté, pour obtenir l'attestation de réussite, 50% des points suffisent. Évidemment, même si ce pourcentage ne sera pas noté sur votre attestation, un meilleur taux de réussite signifiera que vous avez une connaissance plus profonde de la matière.

Ainsi, la profondeur de vos connaissances sera corrélée avec votre choix de suivre le parcours vert, le parcours bleu ou le parcours rouge.

Les parcours vert, bleu ou rouge ne sont que des indications sur la difficulté de chaque exercice proposé. Libre à vous de panacher en faisant les exercices de votre choix.

NOTATION

Le cours comporte des évaluations de votre apprentissage tout au long des différents modules, soit sous forme de quiz soit sous forme d'exercices de codage. Aucune évaluation finale sous forme d'un examen ou quiz de fin de cours n'est donc proposée.

L'évaluation tout au long de l'apprentissage s'effectue par un système classique de points, et votre note sera calculée sur un total de **300 points**, répartis en :

- 102 points sur les **quiz** qui déterminent si vous avez assimilé la matière à raison de **1 point par réponse correcte. Attention vous n'avez généralement qu'un seul essai pour obtenir la bonne réponse ;**
- 150 points sur les **codes UpyLaB** à fournir qui valident votre autonomie dans le codage avec la matière vue jusque-là, à raison généralement de **2 points par exercice UpyLaB noté** qui passe correctement la vérification ;
- 48 points sur le **projet** associé au cours, que nous appellerons projet « Vasarely » et qui est donné en début de module 5 ; les points sont répartis en 24 points provenant des **évaluations par vos pairs** et 24 points provenant d'une **auto évaluation** du projet éventuellement amendé suite aux remarques de vos pairs.

Notez que quelques exercices UpyLaB ne sont pas notés même s'ils participent à votre apprentissage.

REMARQUES IMPORTANTES

- 1) Contrairement aux quiz, les exercices UpyLaB peuvent être testés autant de fois que vous le désirez.
- 2) Pour enregistrer votre réponse dans un exercice UpyLaB vous devez cliquer sur le bouton *Vérifier*, et cela, même si vous savez que votre code n'est pas correct ; ce n'est pas un souci puisque le nombre de fois que vous vérifiez chaque exercice UpyLaB n'est pas limité.
- 3) Attention, si, pour un exercice UpyLaB, la dernière vérification est négative, la plateforme ne retient pas si une vérification précédente avait été positive. En clair, la plateforme ne vous accorde les points que si la **dernière** vérification de cet exercice est positive.

- 4) Si vous décidez de ne pas faire le projet, pour obtenir votre attestation de réussite, vous devrez obtenir 150 points sur les 252 points qui restent en jeu. Libre à vous de faire le parcours vert, mais de quand même décider de faire le projet pour obtenir des points en plus grâce à cela.
- 5) De même ne pas faire certains exercices UpyLaB signifie que sciemment, vous décidez de ne pas avoir de points pour ces exercices. Vous pourrez toujours, avant l'échéance finale du cours, changer d'avis pour améliorer votre score.

ATTESTATION DE SUIVI AVEC SUCCÈS

Tout apprenant atteignant un taux de réussite de 50% des points avant l'échéance recevra une **attestation de suivi avec succès** produite et envoyée par la plateforme FUN.

1.2.4 Naviguer dans notre MOOC

COMMENT NAVIGUER DANS UN MOOC FUN

Un MOOC contient de nombreuses ressources vidéos, textuelles, des exercices à soumettre et à faire viser par la plateforme mais aussi des forums de discussions, des tableaux montrant votre progression, etc.

Il est donc important de bien maîtriser la plateforme FUN pour être à l'aise dans ces aspects pratiques. Pour les débutants sur FUN, un mini cours de démonstration de la plateforme FUN est proposé pour vous y familiariser.

Note : Voir le cours en ligne section 1.2.4 pour accéder au mini cours d'utilisation de la plateforme FUN.

RESSOURCES ET ÉLÉMENTS DE NAVIGATION DE NOTRE MOOC

Si nous parcourons notre MOOC « Apprendre à coder avec Python », les onglets suivants sont présents :

Cours

Il vous donne accès aux 7 modules du cours dont le module « Projet », au fur et à mesure de leur mise à disposition.

Infos du cours

L'équipe pédagogique y communique des nouvelles et informations importantes comme l'ouverture d'un module, un rappel des échéances, ou encore des éventuelles informations sur l'état d'avancement du cours.

Discussions

Vous y trouverez les forums ou espaces de discussion du cours :

- Forum pour faire connaissance en début de cours (module 1) ;
- Forums généraux de chaque module : pour répondre aux questions sur la matière du module ou aux soucis d'accès aux pages ... ;
- Forums des exercices de codage UpyLaB : notre outil que nous présentons dans une section suivante et qui vous permettra de vous exercer ; ces forums sont à utiliser si vous avez des soucis avec un exercice de codage UpyLaB que vous n'arrivez pas à résoudre malgré la consultation du cours et de la FAQ (Foire Aux Questions) sur le sujet ;
- Forum sur le projet (donné en module 5) et son évaluation (réalisée au module « Projet ») ;

Les échanges entre les apprenants et l'équipe pédagogique se font majoritairement via ces forums.

Wiki

Il ne sera utilisé que pour déposer les oeuvres d'art que vous et les autres apprenants aurez produites grâce au programme réalisé en projet.

Progression

Il contient deux parties :

- 1) D'abord un tableau récapitulatif des notes obtenues jusqu'à présent, comme le montre la figure « Tableau récapitulatif de la progression » :

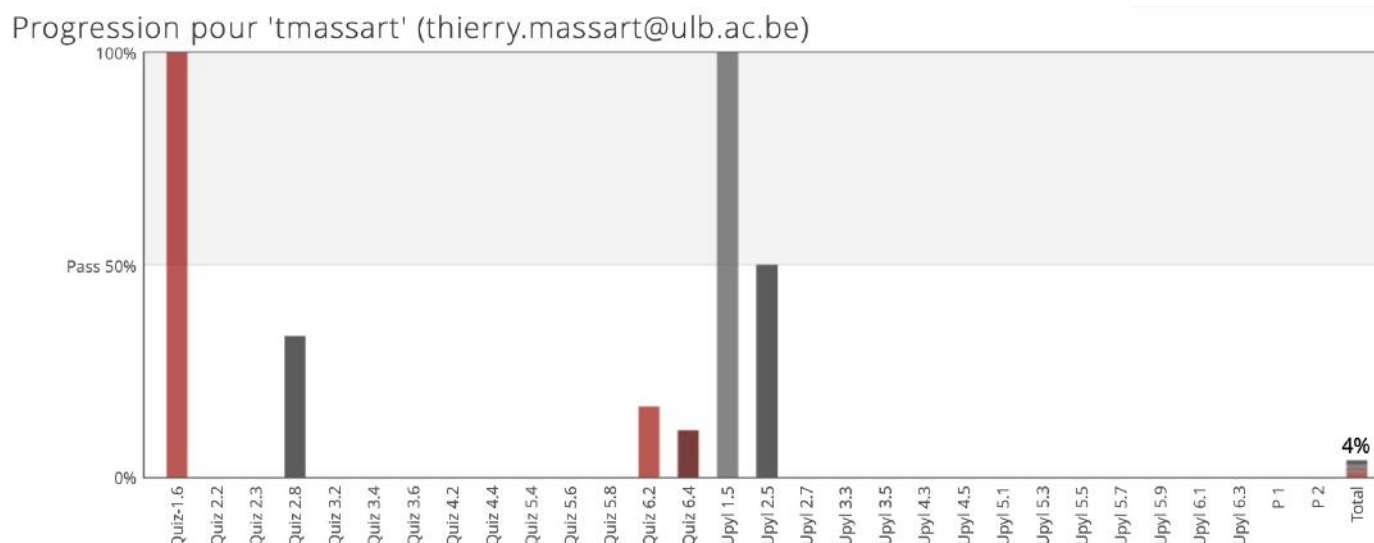


Fig. 1.1 – Tableau récapitulatif de la progression

Ce tableau contient :

- d'abord des colonnes se rapportant aux scores des quiz : avec autant de colonnes que de sections où un quiz est proposé ; par exemple Quiz 2.2 qui correspond aux quiz de la section 2.2 ;
 - ensuite des colonnes se rapportant aux scores des exercices de codage UpyLaB, avec de même autant de colonnes que de sections où un ou des exercices de codage UpyLaB sont proposés ; par exemple : Upyl 2.5 qui correspond aux exercices UpyLaB de la section 2.5 ;
 - deux colonnes pour les notes du projet (évaluation par les pairs et auto-évaluation) ;
 - et enfin le pourcentage approximatif du *Total* obtenu.
- 2) Ensuite la liste des sections avec pour chaque section, les types d'exercices dans cette section et les notes possibles et obtenues jusqu'à présent, comme le montre la figure « Exercices dans chaque section » :

1.2.5 Aperçu de la suite de ce module

APERÇU DE LA SUITE DE CE MODULE

Note : Voir la vidéo de la section 1.2.5 du cours en ligne : Installation de Python 3

CONTENU DE LA VIDÉO

La vidéo précédente présente la suite du module 1 : l'installation de l'interpréteur Python 3 ainsi que de l'environnement de développement PyCharm qui seront utilisés tout au long de ce cours.

1.3 Mode d'emploi pour installer Python 3 et l'environnement PyCharm

1.3.1 Que va-t-on installer ?

Module 1 - Bienvenue dans l'environnement Python 3

Section 1.1 : Première mise en contact

Pas d'exercice noté dans cette section

Section 1.2 : Modalités pratiques du cours

Pas d'exercice noté dans cette section

Section 1.3 : Mise en route de Python 3

Pas d'exercice noté dans cette section

Section 1.4 : Mode d'emploi pour installer Python 3 et l'environnement PyCharm

Pas d'exercice noté dans cette section

Section 1.5 : UpyLaB, Python Tutor et la documentation officielle (2/2) 100%

UpyLaB Section 1.5

Score aux exercices: 2/2

Section 1.6 : Au fait, c'est quoi un code ou un programme ? (3/3) 100%

Quiz Section 1.6

Score aux exercices: 1/1 1/1 1/1

Section 1.7 : Références et bilan du module

Pas d'exercice noté dans cette section

Section 1.8 : Des questions ? Des difficultés ?

Pas d'exercice noté dans cette section

Module 2 - Python comme machine à calculer et à écrire

Section 2.1 : Tout un programme au menu

Pas d'exercice noté dans cette section

Fig. 1.2 – Exercices dans chaque section

INSTALLATION DE PYTHON 3

Cette section vous montre comment installer les deux « outils » qui vont nous servir tout au long du cours :

- **l'interpréteur Python 3** qui permet à notre ordinateur de comprendre le code Python 3, et
- **l'environnement de développement PyCharm** qui va nous permettre
 - d'une part, d'ouvrir des **consoles** pour voir l'effet de chaque instruction Python 3 ;
 - et d'autre part, d'éditer les codes, nommés **scripts** en Python, que nous allons écrire avant de les **exécuter**, terme barbare pour dire que l'on demande à l'ordinateur de réaliser ce que le code demande.

Ces deux installations sont importantes pour ce cours, puisque nous allons utiliser *l'interpréteur Python 3* et *l'environnement PyCharm* de façon intensive.

Donc, quand vous aurez installé Python 3, il faudra encore faire un effort pour installer PyCharm. Plus bas dans cette section, nous expliquons plus en détails pourquoi il faut réaliser ces deux installations. Malgré tout, si l'installation de Python 3 et/ou de Pycharm est impossible ou non désirée sur votre ordinateur, et que vous n'avez pas votre propre solution pour exécuter du code Python 3, nous proposons plus bas dans cette section des solutions alternatives pour pouvoir quand même suivre le cours.

INSTALLATION DE PYCHARM

PyCharm est un environnement de développement pour les programmes Python, plus pompeusement appelé Environnement de Développement Intégré (EDI, ou IDE avec l'acronyme anglais).

Cela signifie que lorsque nous désirons écrire un projet Python, au lieu d'ouvrir un éditeur de texte tel qu'un de ceux proposés classiquement par votre ordinateur, il vaut mieux ouvrir l'application PyCharm qui va s'occuper de vous tout au long de votre travail de programmation : d'abord lors de l'écriture du programme, ensuite, pour l'exécuter pour comprendre ce qu'il fait.

Il existe plusieurs « éditions » de Pycharm. Nous utiliserons l'édition Communautaire (Community en anglais) qui peut déjà faire beaucoup plus que ce dont nous avons besoin pour ce cours et qui est totalement gratuite.

L'installation de PyCharm dépend du système d'exploitation que votre ordinateur utilise.

Avant de procéder à l'installation de l'IDE PyCharm :

- vérifiez que la configuration de votre ordinateur est suffisante. Pour cela, connectez-vous au [site de téléchargement de PyCharm](#)
- cliquez sur le bouton Download
- choisissez en cliquant sur le bouton correspondant au système de votre ordinateur
- et ensuite en cliquant sur « system requirement » pour vérifier que votre ordinateur est suffisant.

Si votre ordinateur n'a pas les caractéristiques suffisantes, vous pouvez soit utiliser l'IDE plus simple IDLE, qui malheureusement n'est pas suffisant pour les codes un peu trop complexes que nous ferons en fin de cours, soit utiliser l'interpréteur distant Trinket via une page web comme expliqué plus bas dans ce module.

Note : Autre IDE :

Il se peut aussi que vous ne débutiez pas totalement et que utilisiez déjà un environnement de développement Python 3 autre que PyCharm Community (par exemple, Thonny, Spyder, Eclipse + Pydev, ...). Dans ce cas, soit vous installez quand même PyCharm pour découvrir ses possibilités et pouvoir interagir avec les autres apprenants, soit vous continuez avec votre environnement, mais dans ce cas, probablement sans que les Foires Aux Questions et forums puissent vous aider en cas de souci avec votre IDE.

1.3.2 Sur mon ordinateur Windows

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3 - EN RÉSUMÉ

Ouvrez un navigateur Web et allez sur le [site officiel de Python](#). Cliquez sur le menu Download -> Python3.x.x (Python 3.6.3 ou ultérieur). Si le site ne propose pas spontanément la bonne version, vous devez sélectionner celle qui convient au système de votre

ordinateur. L'interpréteur Python 3 s'installe en cliquant sur « exécuter » du programme d'installation.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3

Note : Voir la première vidéo de la section 1.3.2 du cours en ligne : Installation de Python 3 sur un ordinateur Windows

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER PYCHARM - EN RÉSUMÉ

- Ouvrons un navigateur web sur le page : <https://www.jetbrains.com/pycharm/>
- Cliquons sur le bouton « Download » qui propose de télécharger PyCharm pour votre système, Windows, MacOS ou Linux dans l'édition Professionnelle ou Community.
- C'est la version Community qu'il nous faut prendre en cliquant sur le bouton download idoine.
- Nous autorisons que le package d'installation (fichier avec le suffixe .exe) s'exécute sur notre ordinateur.
- Nous procédons à son installation en autorisant son exécution et répondant aux questions qu'il pose (y compris si c'est une version 32 ou 64 et si les fichiers .py sont associés à PyCharm - sélectionnez l'option).
- Un petit conseil : installons l'icône PyCharm sur le bureau pour pouvoir ouvrir facilement l'environnement (après l'installation elle se trouve dans les applications JetBrains).

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR WINDOWS

INSTALLER PYCHARM

Note : Voir la seconde vidéo de la section 1.3.2 : Installation de l'environnement PyCharm sur un ordinateur Windows

1.3.3 Sur mon ordinateur MacOS

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3 - EN RÉSUMÉ

Ouvrez un navigateur Web et allez sur le [site officiel de Python](#). Cliquez sur le menu Download -> Python3.x.x (Python 3.6.3 ou ultérieur). Si le site ne propose pas spontanément la bonne version, vous devez sélectionner celle qui convient au système de votre ordinateur. L'interpréteur Python 3 s'installe en cliquant sur « exécuter » du programme d'installation.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3

Note : Voir la première vidéo de la section 1.3.3 : Installation de Python 3 sur un ordinateur MacOS

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER PYCHARM - EN RÉSUMÉ

- Ouvrons un navigateur web sur la page : <https://www.jetbrains.com/pycharm/>
- Cliquons sur le bouton « Download » qui propose de télécharger PyCharm pour votre système, Windows, MacOS ou Linux dans l'édition Professionnelle ou Community
- C'est la version Community qu'il nous faut prendre en cliquant sur le bouton download idoine.
- Nous attendons que le package d'installation soit téléchargé sur notre ordinateur. C'est un fichier avec le suffixe .dmg (ce serait un exe avec le système windows).
- Ensuite procédons à son installation en ouvrant le package d'installation et en plaçant l'application dans le répertoire des applications.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR MACOS

INSTALLER PYCHARM

Note : Voir la seconde vidéo de la section 1.3.3 : Installation de PyCharm sur un ordinateur MacOS

1.3.4 Sur mon ordinateur Ubuntu ou Linux

COMMENT INSTALLER PYTHON 3 ET PYCHARM SUR UBUNTU - EN RÉSUMÉ

Nous allons procéder à l'installation qui vous permettra d'avoir l'interpréteur Python 3 et l'environnement PyCharm. Cette installation est décrite pour la distribution 16.04 du système d'exploitation Ubuntu ; une installation similaire peut être réalisée avec un ordinateur fonctionnant avec un autre système Linux.

Python 3 est déjà installé dans cette distribution, excepté le module turtle qui sera utilisé lors du cours. Pour ajouter ce module utilisez la commande :

```
sudo apt-get install python3-tk
```

Par contre PyCharm n'est pas installé.

COMMENT INSTALLER PYCHARM SUR MON ORDINATEUR UBUNTU - RÉSUMÉ

La procédure est disponible via le site <http://ubuntuhandbook.org> et une recherche sur le mot clé PyCharm.

Ouvrez un terminal (par exemple via Ctrl+Alt+T ou en cherchant et lançant l'application). Dans le terminal exécutez la commande :

```
sudo snap install pycharm-community --classic
```

sudo demande le mot de passe, que vous devez introduire terminé par la touche Enter. Normalement l'installation s'effectue.

La vidéo suivante montre la procédure d'installation et de mise en route de PyCharm.

COMMENT INSTALLER PYCHARM SUR UBUNTU

Note : Voir la vidéo de la section 1.3.4 : Installation de PyCharm sur un ordinateur Linux

1.3.5 Si l'installation ne fonctionne pas

ET SI L'INSTALLATION NE FONCTIONNE PAS

Soit Python 3 et IDLE ont été installés sur votre ordinateur mais ce dernier n'a pas la configuration minimale pour l'installation de PyCharm : dans ce cas,

- soit rabattez-vous sur l'IDE plus simple IDLE ;
- soit utilisez un éditeur simple pour écrire vos scripts et ensuite exécutez-les avec la commande qui invoque l'interpréteur (python3 sur linux ou macOS ou py sur windows) ;
- ou encore exécutez vos codes via le site trinket (voir plus bas).

Soit vous n'avez pas réussi non plus à installer Python 3. Dans ce cas exécutez vos codes via le site Trinket (voir ci-dessous).

Si la configuration de votre ordinateur est a priori suffisante : essayez de voir si le sujet existe dans la FAQ ou sinon mettez à profit l'entraide sur le forum du module, en décrivant précisément le souci et en demandant aux autres inscrits de vous aider. Si malgré tout, cela ne fonctionne pas, utilisez IDLE ou Trinket.

ENTRAIDE LORS DE L'INSTALLATION DE PYTHON 3

Notre cours en ligne est aussi un endroit (virtuel) où l'entraide est la règle d'or. Le forum du module, en fin de module, permet de mettre en contact les apprenants (voir règles de bonnes pratiques des forums via le lien <https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@BonnesPratiquesForum.pdf>).

En particulier dans l'activité précédente :

- Si vous avez rencontré des soucis dans l'installation de l'environnement Python, décrivez le plus clairement possible le problème et précisez **surtout** le système d'exploitation qu'utilise votre ordinateur (exemple : Windows 10 version 1703 ou MacOS 10.12.4...) : notre équipe ou d'autres apprenants qui ont la solution pourront probablement vous aider.
- Si vous avez réussi l'installation après avoir résolu quelques soucis, merci de partager votre expérience en détaillant le système d'exploitation utilisé par votre ordinateur, le souci rencontré et sa résolution.
- Si vous avez la solution à un problème évoqué dans le forum, aider ceux qui l'évoquent en leur fournissant une réponse leur sera d'un grand secours et fera avancer la communauté formée par l'ensemble des participants de notre cours en ligne.

TRINKET SI VOUS N'AVEZ PAS RÉUSSI À INSTALLER PYTHON 3

Trinket permet d'exécuter du code Python sur une plateforme distante, c'est-à-dire sans que l'interpréteur Python ou un IDE local soit installé sur son ordinateur.

Trinket est une bonne alternative pour apprendre Python 3 même si vous n'aurez pas la flexibilité offerte par un environnement complet comme fourni par PyCharm.

Il suffit de se connecter à l'adresse <https://trinket.io> et de créer un compte en cliquant sur l'onglet Sign up et en remplissant le formulaire (votre nom, un nom d'utilisateur, votre adresse email et un mot de passe).

Une fois votre compte créé, vous pouvez vous connecter et trouver l'interpréteur Python 3 par exemple en cliquant sur le bouton new trinket -> Python 3.

Vous pouvez dès lors taper votre code dans la fenêtre main.py : la première ligne du code doit toujours être la suivante :

```
#!/bin/python3
```

qui indique à Trinket que la suite est du Python 3 (avec la convention Unix).

Par exemple :

```
#!/bin/python3
print('Hello World')
```

et ensuite vous pouvez demander l'exécution du code en cliquant sur le triangle au dessus de la fenêtre `main.py`

1.3.6 Installation d'un dictionnaire français dans PyCharm

L'environnement de développement PyCharm est d'une grande aide aux codeurs. En particulier, il vérifie si ce que nous encodons semble être du code Python cohérent. Il nous mentionne également si nous utilisons dans nos explications (nous verrons qu'elles s'appellent commentaires) des mots corrects. Pour cela PyCharm utilise un ou des dictionnaires.

Pour nous aider, nous proposons d'installer le dictionnaire des mots français (y compris, nous verrons pourquoi, les mots dont les accents ont été supprimés). Avant de procéder à l'installation elle-même, nous devons télécharger ces deux dictionnaires.

TÉLÉCHARGEMENT DES FICHIERS

Vous pouvez réaliser ce téléchargement :

- `dictionnaire_mots_français` via le lien [mots](#)
- `dictionnaire_français_avec_et_sans_accents` via le lien [mots_avec_et_sans_accents](#)

Maintenant que les dictionnaires sont téléchargés sur votre ordinateur (assurez-vous de savoir où ils se trouvent), renommez les `mots.dic` et `mots_avec_et_sans_accents.dic` et voyons la procédure pour installer ces dictionnaires dans l'environnement PyCharm.

INSTALLATION DANS PYCHARM

- Après avoir ouvert PyCharm, cliquez sur le menu `File` ensuite `Settings` (ou `PyCharm` ensuite `Preferences` suivant le système que vous utilisez (Windows, MacOS, ...)) (voir figure « Installation dictionnaire étape 1 »);
- cliquez sur le petit triangle devant l'intitulé `Editor` pour ouvrir ce sous-menu (voir figure « Installation dictionnaire étape 2 »);
- ensuite cliquez sur `spelling` (voir figure « Installation dictionnaire étape 3 »);
- dans la sous-fenêtre `Custom Dictionaries` cliquez sur le bouton `+` (voir figure « Installation dictionnaire étape 4 »);
- et trouvez et sélectionnez le dictionnaire à ajouter en cliquant sur `open`;
- le dictionnaire s'ajoute à la liste des dictionnaires (voyez les deux nouvelles coches) (voir figure « Installation dictionnaire étape 5 »);
- sortez du menu en cliquant sur le bouton `ok`.

UTILISER PYCHARM

1.3.7 Utilisation de PyCharm

Vous avez installé PyCharm. Lancez-le et créez un nouveau projet. Voyons comment créer et exécuter un script.

Un projet peut contenir plusieurs scripts bien sûr. Soit qui sont liés (c'est le cas dans les gros projets réels) soit qui sont indépendants mais appartiennent à une même famille... C'est le cas par exemple si je décide de faire un projet par Module de mon MOOC :

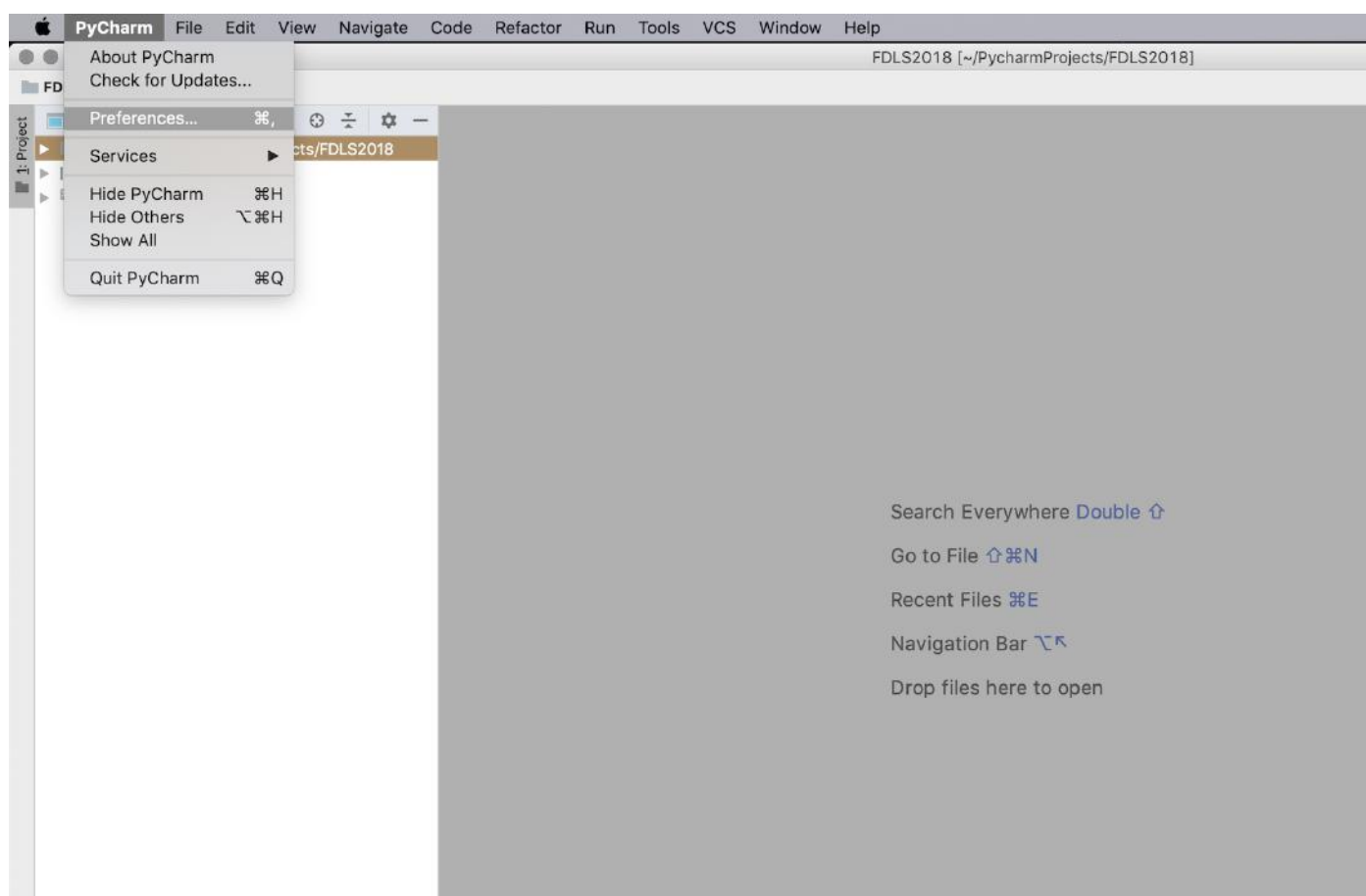


Fig. 1.3 – Installation dictionnaire étape 1

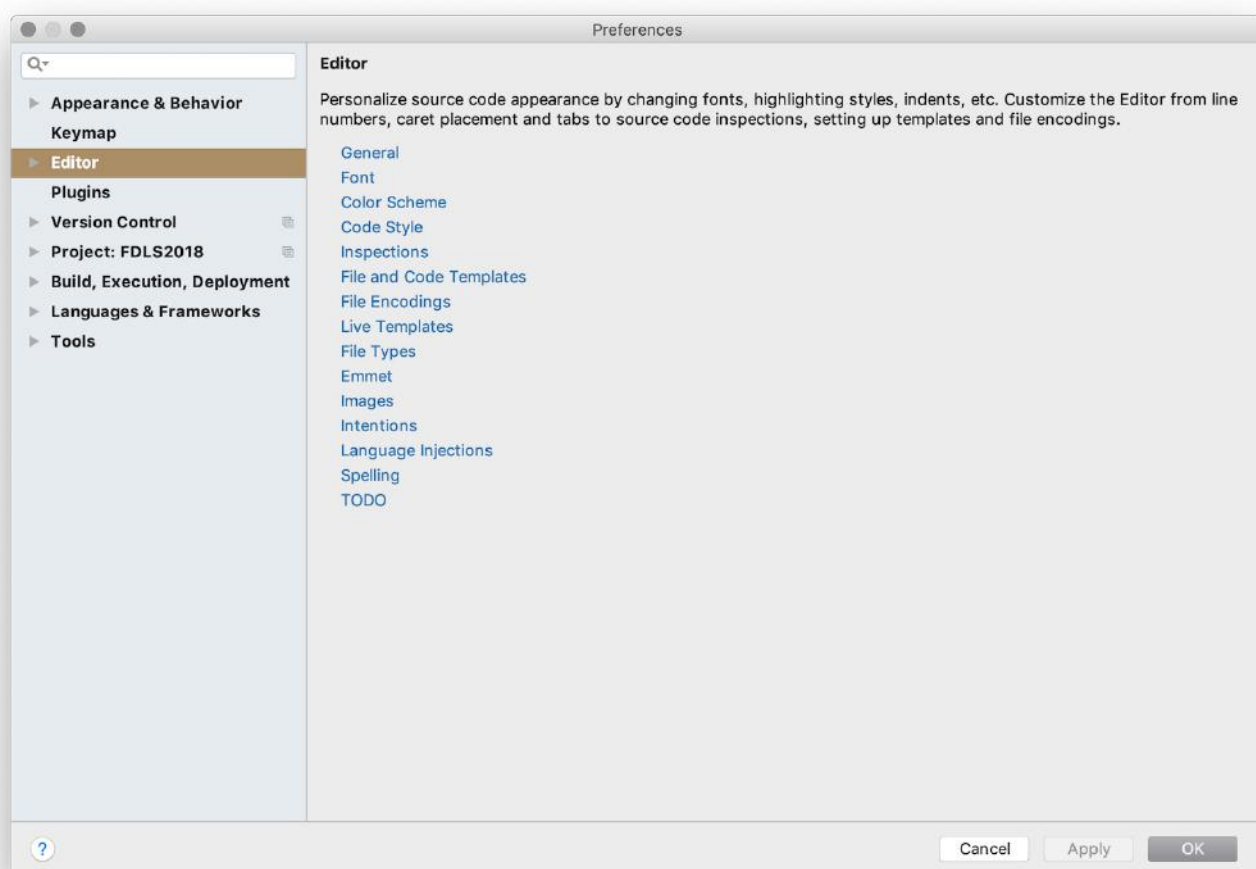


Fig. 1.4 – Installation dictionnaire étape 2

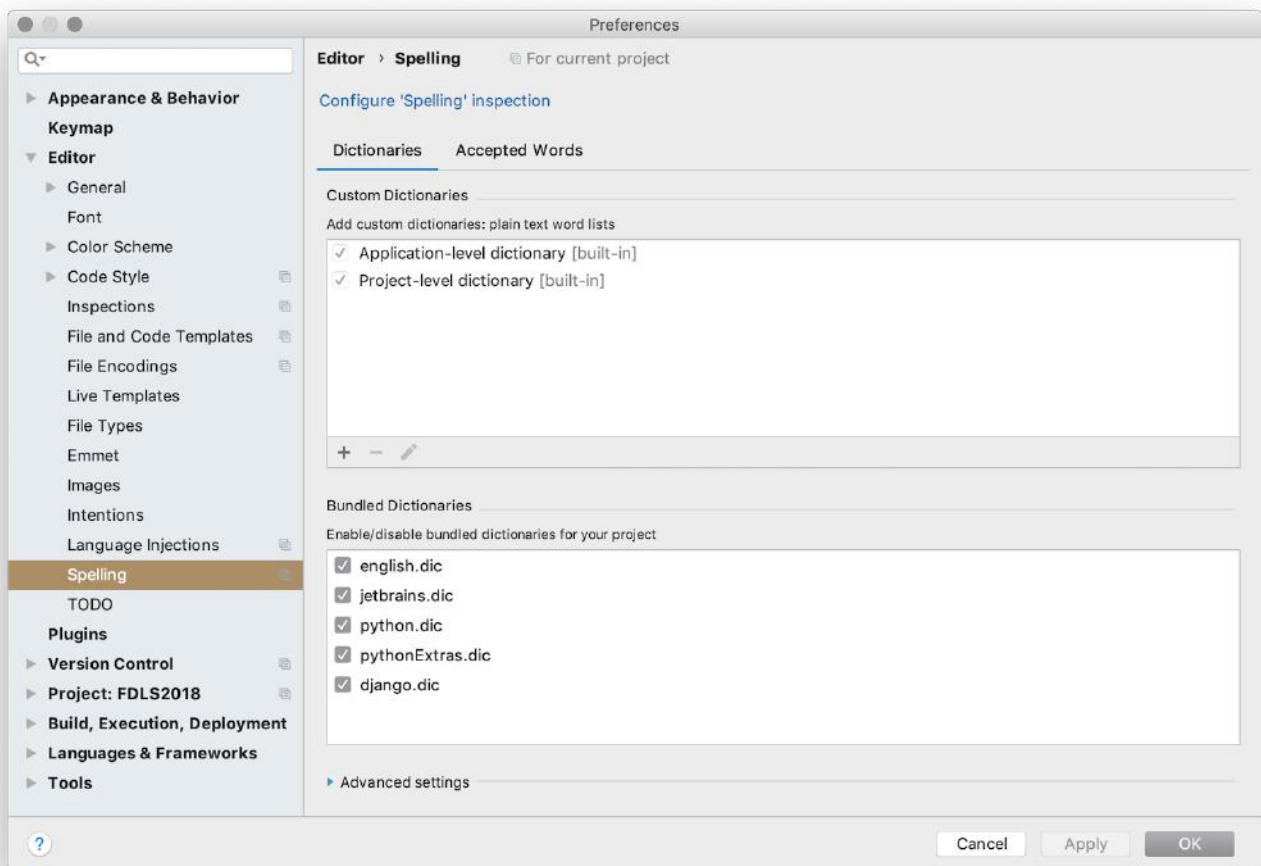


Fig. 1.5 – Installation dictionnaire étape 3

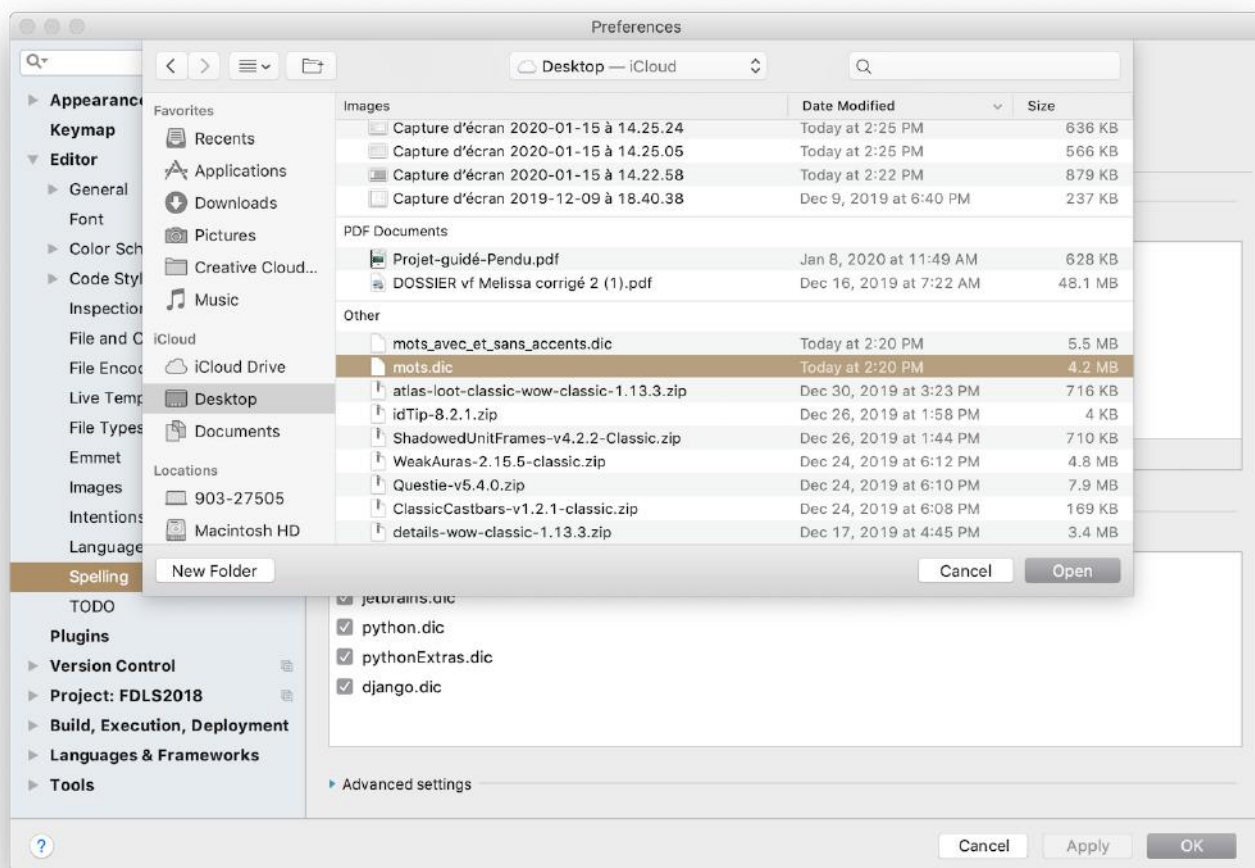


Fig. 1.6 – Installation dictionnaire étape 4

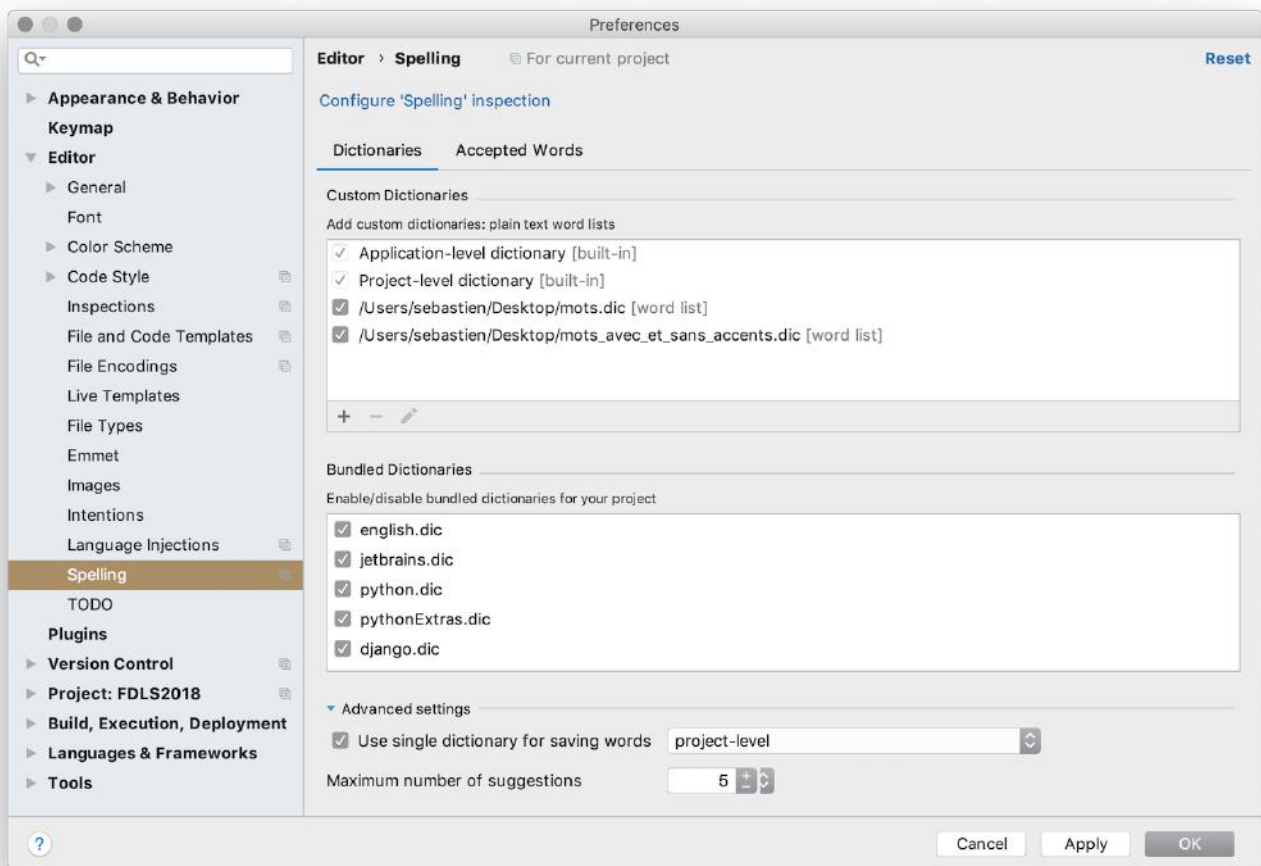


Fig. 1.7 – Installation dictionnaire étape 5

je mettrai dans le projet `Module_1` tous mes exercices du Module 1, etc. On peut aussi faire un gros projet `MOOC_Python` et y mettre tous ses scripts sans distinction de module.

Cette organisation est bien sûr question de goût, d'habitude de travail et donc propre à chacun.

Lorsque votre projet est créé, pour y ajouter un nouveau script vous pouvez utiliser un clic droit sur le nom du projet, dans la colonne de gauche, et faire `New > Python File`. Vous donnez un nom à ce script qui vient se rajouter aux autres. Dans la colonne de gauche vous avez l'ensemble des scripts. Ainsi, les deux captures suivantes montrent comment j'ai créé un nouveau script nommé `test_2.py` (voir les deux figures « Création d'un nouveau script dans PyCharm »).

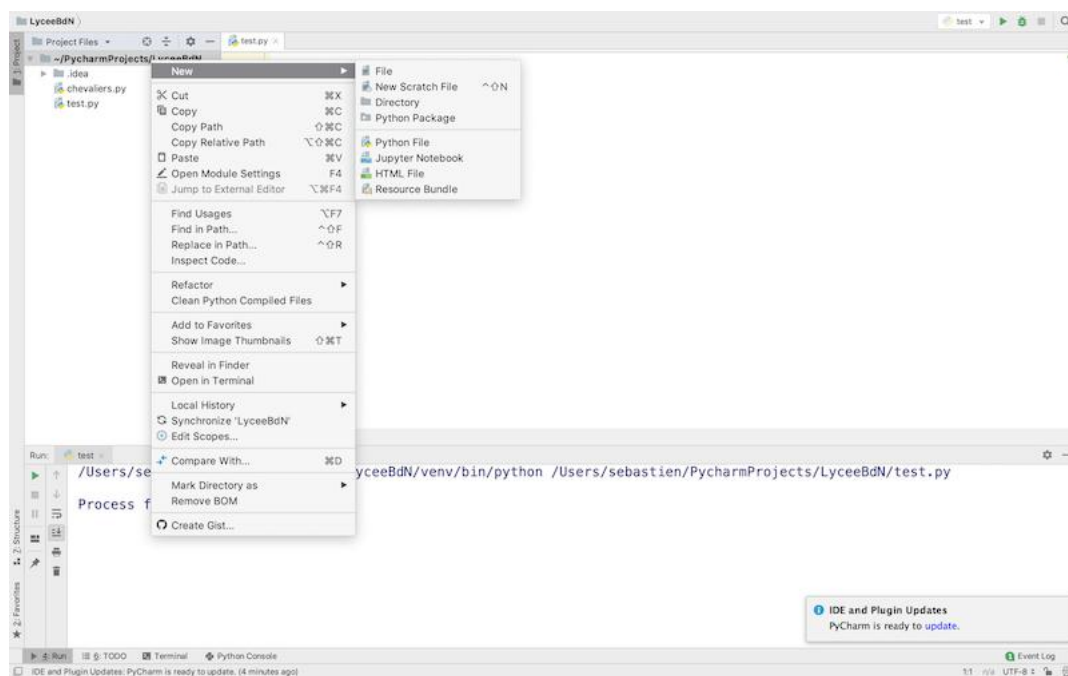


Fig. 1.8 – Création d'un nouveau script dans PyCharm

Parmi tous les scripts de votre projet, pour en exécuter un, il y a deux cas de figure :

- 1) soit il est sélectionné comme script courant (il apparaît en haut à droite avec la petite flèche verte) auquel cas utiliser cette petite flèche ou faire `Run` dans le menu exécutera ce script ;
- 2) soit il ne l'est pas et alors le plus simple est de faire, dans la colonne de gauche, un clic droit sur le nom du script que vous voulez exécuter (disons `test_2.py` comme sur la capture ci-dessous) et choisir dans le menu contextuel `Run test_2` (voir figure « Exécution d'un script `test_2` dans PyCharm ») :

Et les scratch file ?

Les scratch files sont des fichiers python comme les autres mais numérotés automatiquement et rangés dans un dossier (projet) à part. Par exemple ci-dessous j'ai créé un nouveau scratch file, (ce n'est pas mon premier comme l'atteste le 2 dans le nom) et il n'apparaît pas dans mon Projet (qu'on voit à gauche). Mais si je fais apparaître le menu contextuel en cliquant-droit sur le nom `scratch_2.py`, je vois que j'ai un `Run scratch_2` et je peux donc l'exécuter (voir figure « Exécution d'un script dans un fichier scratch dans PyCharm »).

Conseil

De préférence, créez des fichiers python dans votre ou vos projets et n'utilisez pas les *scratch files* de peur de ne plus savoir où se trouvent vos codes.

1.4 UpyLaB, Python Tutor et la documentation officielle

1.4.1 Notre exerciceur UpyLaB

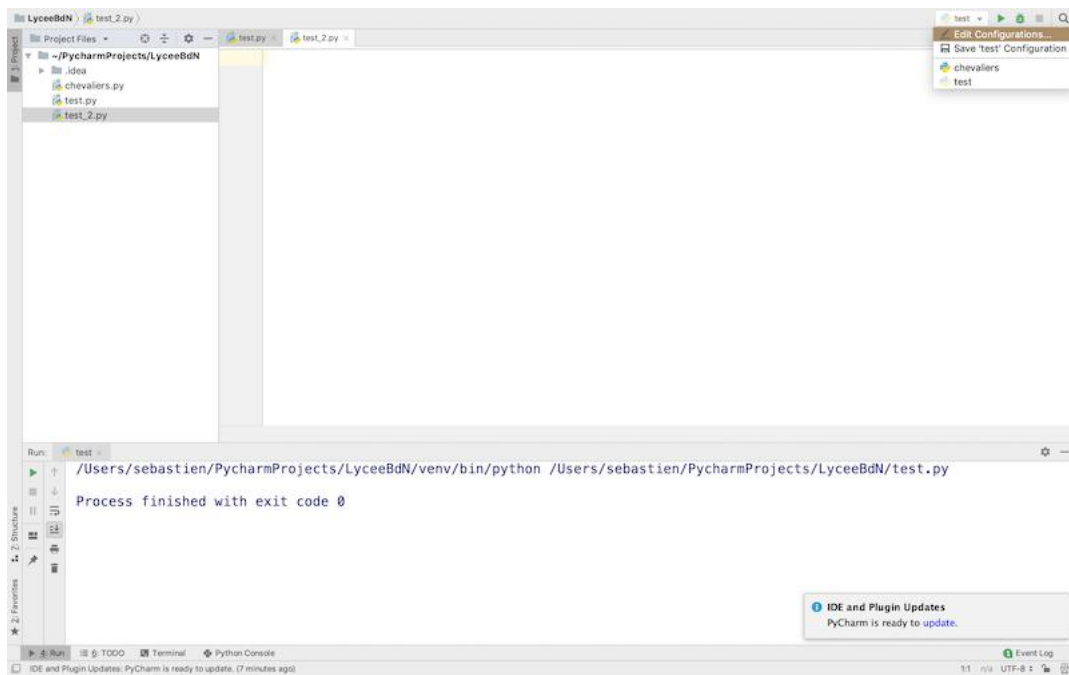


Fig. 1.9 – Création d'un nouveau script dans PyCharm (suite)

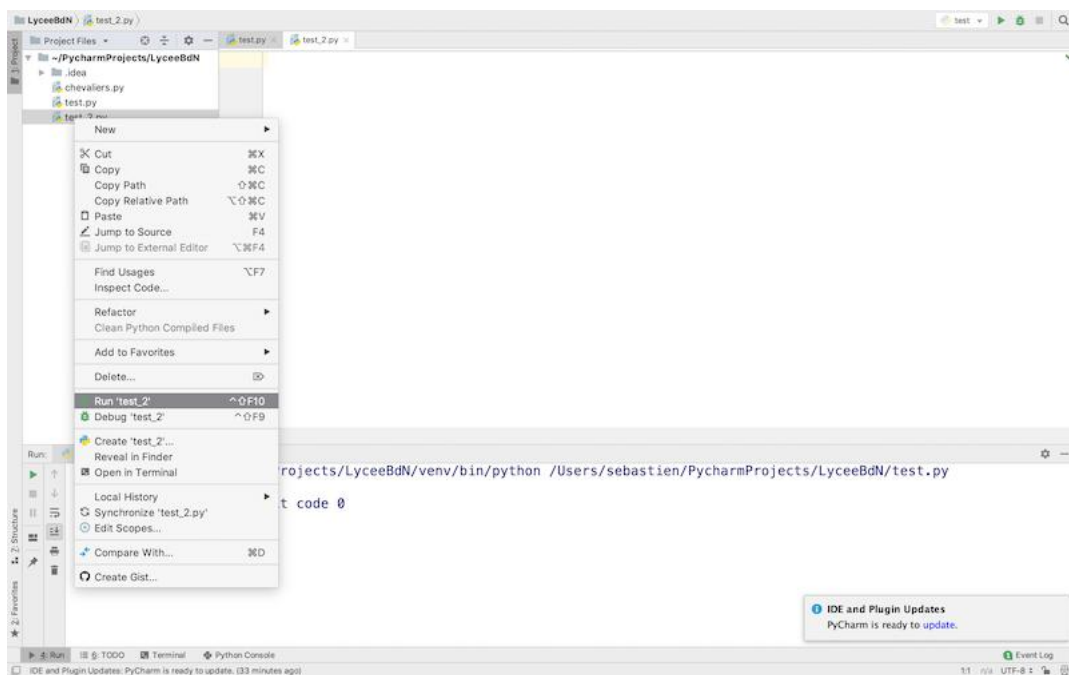


Fig. 1.10 – Exécution d'un script test_2 dans PyCharm

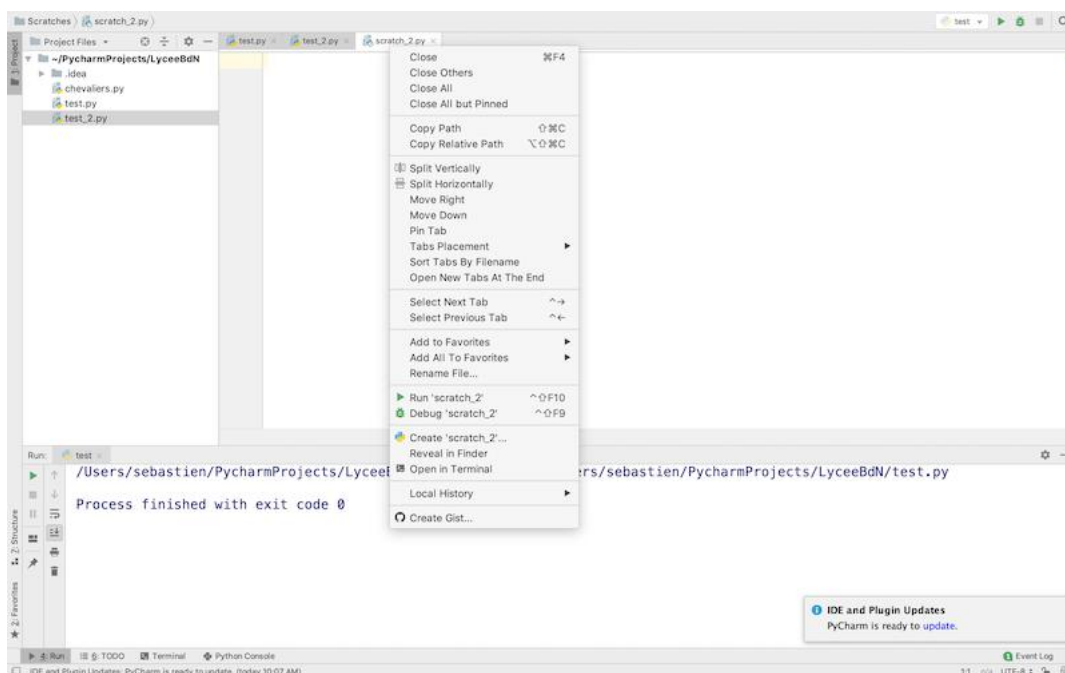


Fig. 1.11 – Exécution d'un script dans un fichier scratch dans PyCharm

UPYLAB

Dans ce cours, en plus de l'interpréteur Python 3 et de l'environnement de développement PyCharm, nous utilisons deux « outils » : **UpyLaB** et **Python Tutor**. Présentons d'abord UpyLaB.

UpyLaB est notre plateforme d'apprentissage en ligne. Elle propose des exercices de codage et demande aux étudiants de produire le code Python correspondant ; ensuite UpyLaB, qui est exécuté sur un serveur extérieur à la plateforme FUN, teste si la solution proposée est correcte. Nous avons intégré l'utilisation d'UpyLaB au cours en ligne, comme le montre le premier exercice UpyLaB proposé à la page suivante.

TESTER UPYLAB

L'exercice UpyLaB 1.1 ci-après vous permet de tester l'utilisation d'UpyLaB ; en clair voir comment on l'utilise dans ce cours. Le principe est de lire l'énoncé, de résoudre le problème (conseil : d'abord, dans l'environnement PyCharm). Ici on vous demande juste d'écrire l'instruction Python 3 (c'est-à-dire l'ordre que l'on donne à l'ordinateur) :

```
print("Bonjour UpyLaB !")
```

et ensuite de copier/coller **votre code qui solutionne l'exercice** dans la partie de fenêtre UpyLaB servant à cet effet (mini-fenêtre vide au départ, sinon qui contient votre dernière proposition de solution) et de vérifier que votre code solutionne bien le problème en cliquant sur le bouton « Vérifier ».

Nous n'avons pas encore vu l'effet de l'instruction `print`. Actuellement, vous devez juste comprendre que UpyLaB, quand l'utilisateur a cliqué le bouton « Vérifier », va demander à l'interpréteur Python 3 d'exécuter le code fourni. Ensuite, il va comparer le résultat avec ce qui est attendu et valider le résultat s'il coïncide.

UpyLaB effectue un ou plusieurs tests et ensuite, en fonction de cela, vous indique si le code soumis lui semble correct ou non par rapport à ce qui vous est demandé.

Comme pour l'installation de Python 3, en cas de besoin, nous vous proposons d'utiliser la FAQ et si vos soucis ne sont pas résolus, le forum UpyLaB du présent module afin de vous mettre en contact avec les autres apprenants et l'animateur des forum.

En particulier, si vous rencontrez des soucis pour réaliser l'exercice 1.1 d'UpyLaB, décrivez le plus clairement possible le problème : notre équipe ou d'autres apprenants plus expérimentés pourront probablement vous aider.

1.4.2 Exercice UpyLaB 1.1 - Parcours Vert, Bleu et Rouge

TESTER UPYLAB EN FAISANT L'EXERCICE UPYLAB 1.1

Les seuls buts de cet exercice sont de vérifier que vous avez accès à notre exerciceur UpyLaB et de vous donner un premier contact avec cet outil.

Écrire un programme qui affiche « Bonjour UpyLaB ! » grâce à l'instruction :

```
print("Bonjour UpyLaB !")
```

Veillez à ne pas ajouter d'espace au début de la ligne et à parfaitement respecter les espaces, minuscules et les majuscules (ici pourquoi ne pas simplement faire une copier/coller de l'instruction dans la fenêtre UpyLaB)

Pour tous les exercices UpyLaB, vous devez écrire **le code Python qui solutionne le problème (c'est-à-dire votre programme Python)** dans la sous-fenêtre intitulée « Entrez votre solution ».

Exemple

Lorsqu'il est exécuté, le programme affiche :

```
Bonjour UpyLaB !
```

Bien sûr, libre à vous d'être curieux et :

- de voir ce que répond UpyLaB quand vous donnez une autre réponse, par exemple :

```
print("Bonsoir UpyLaB !")
```

ou en ajoutant une ou plusieurs espaces devant l'instruction ;

- même si l'instruction `print` ne vous a pas encore été présentée, de comprendre sa traduction française pour avoir une idée de ce qu'elle pourrait bien avoir comme effet.

Note : en cas de besoin, la section *PROCEDURE POUR UTILISER UPYLAB* ci-dessous peut solutionner vos problèmes ou répondre à plusieurs de vos questions.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Vous obtenez une erreur de syntaxe ? : Vérifiez que vous avez bien copié l'instruction demandée, sans ajouter d'espace devant, et sans oublier parenthèses ou guillemets fermants.
- Pas d'erreur de syntaxe, mais UpyLaB refuse de valider votre code ? : L'affichage doit être exactement identique à celui attendu. Veillez en particulier à respecter majuscules / minuscules, à ne pas ajouter ou ôter des espaces (en particulier à la fin de la phrase), à ne pas oublier la ponctuation.
- Un conseil pour avoir un message identique est de le copier depuis l'énoncé pour le coller dans votre code. Ici, vous pouvez même copier toute l'instruction :

```
print("Bonjour UpyLaB !")
```

avant de la coller dans la fenêtre UpyLaB de l'exercice.

- Si rien ne marche : consultez la *FAQ sur UpyLaB 1.1*.

PROCEDURE POUR UTILISER UPYLAB

Comme expliqué, chaque fenêtre UpyLaB donne un exercice à résoudre sous forme de code Python. Une procédure d'identification est mise en place entre les pages FUN et UpyLaB. Voici la procédure précise qui inclut une procédure pour rafraîchir la page web si un certain délai est passé après cette identification.

Procédure :

- 1) Après avoir développé dans l'environnement de développement PyCharm le code demandé,
- 2) copiez ce code pour le coller dans la sous-fenêtre UpyLaB correspondante ;
- 3) cliquer sur le bouton « Vérifier ».
- 4) Si le code passe les tests, félicitations vous obtenez les points correspondants.
- 5) Si les tests ne sont pas concluants, vous devez corriger votre code et recommencer la procédure.
- 6) Si par contre un message d'erreur apparaît (Erreur serveur : session expirée ou non-trouvée. Veuillez rafraîchir la page dans le navigateur),
 - cliquez sur le bouton Rafraîchir la page si demandé par UpyLaB avant de vérifier à nouveau
 - et ensuite cliquez à nouveau sur le bouton Vérifier
 - et reprenez la procédure au point 4.

La fenêtre UpyLaB reste bloquée sur Connexion en cours ...

Pour certains navigateurs Web et certaines configurations, des problèmes d'accès à notre outil UpyLaB peuvent survenir. Typiquement, la fenêtre reste bloquée avec un message `Connexion en cours ...`

Si cela se produit, cliquez simplement sur le bouton Rafraîchir la page si demandé par UpyLaB avant de vérifier à nouveau situé en dessous de l'exercice (même si ce n'est pas demandé explicitement :-)).

Si le souci perdure, une des causes les plus courantes est l'utilisation de code anti « pop-up » ou anti-mouchard (en particulier avec Firefox). Si c'est le cas, il est possible que vous deviez autoriser l'accès à <https://upylab.ulb.ac.be> (l'adresse d'accès à notre outil UpyLaB).

Donc si une telle erreur se produit, allez voir dans la FAQ (Foire Aux Questions) (onglet FAQ) ou cliquez sur l'onglet Discussion et trouvez le fil de discussion Problèmes d'accès à UpyLaB où un résumé des soucis constatés et des solutions précises proposées y a été déposé.

COMMENT RÉSOUDRE LES EXERCICES UPYLAB

Un point essentiel à fixer pour bien apprendre lors de ce cours est la façon de réaliser les exercices UpyLaB. Pour chaque exercice UpyLaB, l'énoncé est fourni.

UpyLaB n'est pas un environnement de développement. Il permet juste de tester si votre code lui semble correct.

Ainsi, après avoir bien compris l'énoncé de l'exercice, il est important de développer une solution en utilisant l'IDE PyCharm (ou si PyCharm n'est pas disponible, IDLE ou Trinket) ou, pour voir de façon détaillée comment votre code s'exécute, l'outil Python Tutor que nous présentons à la section suivante. Ce n'est que quand votre solution sera complète et que vous l'aurez validée en testant le code que vous pourrez copier et coller ce code dans la fenêtre UpyLaB de l'exercice pour lancer la validation. Tester son code signifie exécuter le programme plusieurs fois et, si possible, sur des exemples différents, pour être convaincu qu'il donne toujours une réponse correcte.

Si UpyLaB ne le valide pas - j'ai par exemple mis l'instruction :

```
print("Bonsoir UpyLaB !")
```

ou même

```
print("Bonjour UpyLaB ! ")
```

avec **une espace en trop après le point d'exclamation**,

à la place de :


```
print("Bonjour UpyLaB !")
```

vous devez le corriger avant de soumettre une solution modifiée.

Attention : Le nombre de vérifications que vous pouvez faire avec UpyLaB n'est pas limité. Malgré tout, il est préférable de réussir l'exercice avec un nombre de clics « Vérifier » le plus petit possible.

Nous insistons sur le fait qu'UpyLaB est un exerciceur avec un environnement de tests et non un environnement de développement. Prenez donc l'habitude de développer chacun de vos codes dans un environnement de développement comme PyCharm, et ensuite, quand vous le jugez correct et complet, de faire un copier-coller dans la fenêtre UpyLaB qui doit recevoir le code qui solutionne le problème, pour ensuite cliquer sur le bouton « Vérifier » afin de réaliser les tests.

1.4.3 L'outil Python Tutor

PYTHON TUTOR

Python Tutor est un outil en ligne créé et maintenu par Philip Guo (<http://www.pgbovine.net>) de l'Université de San Diego. Comme tout ce que nous utilisons pour ce cours, son utilisation est totalement libre et gratuite.

Python Tutor permet d'exécuter pas à pas des petits scripts Python en visualisant l'effet de chaque instruction Python exécutée, comme le montre le petit exemple ci-dessous. Dans notre cours, nous utiliserons cet outil **de façon intégrée** à nos pages.

Exemple Python Tutor du code Python qui affiche les trois lignes suivantes avant de se terminer

```
Hello World
Bonjour le Monde !
J'apprends Python 3
```

Python Tutor va nous permettre de détailler, instruction par instruction, comment Python fonctionne en cliquant sur les boutons « Forward » (et « Back » pour revenir en arrière) ou en actionnant le curseur.

N'hésitez pas à animer l'exemple ci-dessous en cliquant sur ces boutons et curseur !

Exemple Python Tutor intégré

```
print('Hello World')
print('Bonjour le Monde !')
print("J'apprends Python 3")
```

Note : Voir animation du code avec Python Tutor dans le cours en ligne

Utilisation de Python Tutor

Si, durant ce cours ou plus tard, vous désirez bien comprendre un code Python, vous pouvez accéder à l'outil Python Tutor en ligne via la page <http://pythontutor.com> pour l'utiliser quand cela vous semble utile. Attendons d'avoir vu plus de concepts de Python, au module suivant, pour en faire une démonstration plus complète et bien comprendre son utilité pour l'apprentissage du langage.

Pour l'instant ces concepts sont probablement assez flous pour vous. Ils deviendront plus clairs dès le module suivant quand nous commencerons à expliquer les instructions de base.

1.4.4 La documentation officielle sur python.org

DOCUMENTATION OFFICIELLE

Dans le cadre de ce cours, vous n'aurez normalement pas besoin de consulter la documentation « officielle » Python. En effet la documentation officielle d'un langage de programmation est souvent un peu rédhibitoire pour des débutants. Malheureusement, parfois, c'est le seul endroit où l'on trouve ce que l'on cherche.

L'accès à cette documentation se fait via le site officiel de Python (<https://python.org>) qui contient un menu « documentation » : cliquez sur le bouton « Python 3.x Docs » qui vous propose toute la documentation officielle Python 3, y compris une possibilité en haut à droite de faire une recherche rapide. Essayez cette recherche, et vous verrez qu'au début vous serez probablement noyé dans l'abondance d'information. Ne vous en faites pas pour cela !

1.5 Au fait, c'est quoi un code ou un programme ?

1.5.1 Quelques définitions

INFORMATION, DONNÉE, ALGORITHME : QU'ES AQUO ?

Il est important pour tout un chacun de connaître quelques mots de « jargon » et quelques définitions dans le domaine. Cela permet de mieux comprendre et connaître les informaticiens et leur travail, mais aussi de comprendre leur façon de s'exprimer par exemple lorsqu'ils décrivent un problème à résoudre.

Commençons par brièvement parler de l'informatique. Le but de l'informatique est d'effectuer du **traitement** automatisé de l'**information**.

L'**information** est un ensemble d'éléments qui ont une signification dans le contexte étudié.

Les **données** d'un problème sont représentées par l'ensemble des informations utilisées pour résoudre ce problème en vue d'obtenir les **résultats** escomptés. Pour cela, l'informaticien peut commencer par écrire des algorithmes.

Un **algorithme** n'est pas conçu uniquement pour obtenir un résultat pour une donnée bien précise, mais constitue une **méthode** qui permet, à partir de n'importe quelle autre donnée du même type, d'obtenir le résultat correspondant.

Un exemple simple d'algorithme est celui qui consiste, depuis l'entrée, à trouver la sortie d'un labyrinthe.

- Les données de l'algorithme sont le plan du labyrinthe avec en particulier l'endroit où se trouvent l'entrée et la sortie.
- L'algorithme va consister à entrer dans le labyrinthe et ensuite longer le côté gauche (ou le droit mais sans alterner) et avancer tant que possible, en faisant demi-tour quand nous sommes bloqués mais en continuant à longer le côté gauche. Si le labyrinthe n'a qu'une seule entrée et une sortie sur les côtés extérieurs de celui-ci et qu'il n'a ni pont ni tunnel, cet algorithme permet de trouver la sortie.
- Le résultat de cet algorithme sera par exemple la séquence de mouvements à réaliser pour, depuis l'entrée, trouver la sortie.

Un algorithme peut être **implémenté** sur un ordinateur. Celui-ci ne possède jamais qu'une quantité limitée de mémoire de stockage d'information dont la précision est limitée. De ce fait pour résoudre certains problèmes qui, en théorie, pourraient requérir un calcul trop long, ou une précision ou un stockage d'information trop important, des algorithmes ne donnant qu'une valeur approchée du résultat doivent être conçus.

Dans le contexte de ce cours, on parle de code et de programme informatique sous forme de séquence d'instructions Python 3.

Un **code** est un programme informatique, une partie de programme, ou une liste d'instructions ou de commandes (un « script ») pour le système de votre ordinateur.

Un programme est un ensemble d'instructions (donc du code) qui, quand il est exécuté sur un ordinateur, réalise un traitement défini. Un programme est donc vu comme la traduction en un code compréhensible par l'ordinateur d'un algorithme qui solutionne un problème.

Même si on peut faire de longs débats pour savoir si un bout de code peut être défini comme un programme, et plus important, pour savoir si quelqu'un peut être décrit comme un codeur ou un programmeur (faites par exemple une recherche sur le Web : « différence codeur programmeur »), dans notre contexte, nous supposons que c'est presque la même chose.

Un langage de programmation comme le langage Python définit les règles nécessaires pour le code ou le programme pour qu'il soit compréhensible et exécutable par un ordinateur.

Un programme appelé interpréteur ou compilateur « traduit » ce code source, c'est-à-dire dans le langage de programmation, en code machine. L'interpréteur « exécute » immédiatement chaque instruction analysée, tandis que le compilateur traduit d'abord complètement le code source en code machine qui pourra par la suite être exécuté.

Dans le contexte Python, vous pouvez voir une instruction comme un ordre que l'interpréteur Python donne à l'ordinateur qui exécute le code.

Écrire un code ou un programme ou a fortiori développer un gros logiciel demande une démarche en plusieurs étapes, appelée processus de développement d'un programme, qui peut être divisée en plusieurs phases (partiellement) successives.

- Analyse et spécification de ce qui est requis
- Conception
- Implémentation
- Tests et installation
- Exploitation et maintenance

Chaque phase produit des résultats écrits : spécification de ce qui est requis (cahier de charges), manuel utilisateur, description du fonctionnement, description succincte ou détaillée de l'algorithme, programme dûment commenté, historique des modifications,

Le travail d'un ingénieur système est de mener ces phases ou de les superviser.

Dans ce cours nous nous concentrons sur l'implémentation, c'est-à-dire le codage qui, bien sûr, demande de travailler sur les quatre premières phases de développement logiciel.

1.5.2 Quiz de fin de module

QUIZ DE FIN DE MODULE

Tous les quiz de la formation comptent pour la note finale du cours en ligne. Attention, pour certains quiz, comme celui-ci, vous aurez droit à plusieurs essais, mais pour la plupart, vous n'aurez droit qu'à un seul essai !

Note : Voir le quiz en section 1.5.2 du cours en ligne

POUR FINIR UN PETIT SONDAGE SUR L'INSTALLATION DE PYTHON

Note : Sondage sur l'installation de Python 3 et de PyCharm Community : voir en section 1.5.2 du cours en ligne

Retour d'information

- Si vos installations de Python 3 et PyCharm ont été **totale**ment aisées : chouette on peut y aller ; merci de voir dans le forum si vous pouvez aider un apprenant qui aurait un souci.
- Si vos installations de Python 3 et PyCharm ont été **globale**ment aisées : nous espérons que cela ne vous a pas pris trop de temps, on continue ; merci de voir dans le forum si vous pouvez aider un apprenant qui aurait le même type de souci que celui que vous avez expérimenté.
- Si votre installation de Python 3 et de PyCharm ont été **difficiles** : nous espérons que cela ne vous a pas pris trop de temps, on continue ; merci éventuellement de dire dans le forum comment votre souci a été résolu.
- Si **vous n'êtes pas arrivé** à installer Python 3 ou PyCharm : si personne de votre entourage ne peut vous aider, utilisez le forum ; nous espérons que quelqu'un pourra y solutionner vos soucis.

1.6 Références et bilan du module

1.6.1 Références et bilan

RÉFÉRENCES

Dans ce module nous avons planté le décor. Vous avez installé l'environnement Python 3 avec PyCharm sur votre ordinateur et pris connaissance d'outils qui vont nous aider tout au long du cours ; les références qui suivent complètent la panoplie d'outils.

Références :

- <http://pythontutor.com> : accès à “Start visualizing your code now” qui vous permet de visualiser comment s'exécute pas à pas votre code (ne pas oublier de mettre l'option python 3.6 ou ultérieur)
- Si vous désirez avoir un livre complet en plus du support de cours, le livre de Gérard Swinnen [Apprendre à programmer avec Python 3](#) est une excellente référence pour débiter votre apprentissage
- <https://www.python.org> : site officiel de Python (téléchargements et documentations Python 3)

BILAN DU MODULE

Note : Voir la vidéo de la section 1.6.1 du cours en ligne

RÉSUMÉ DE LA VIDÉO

- Vous avez installé et fait connaissance avec Python 3 ainsi que l'environnement PyCharm.
- Vous avez vu comment soumettre un exercice UpyLaB et comment manipuler une démonstration Python Tutor.
- En route vers le module 2 et l'apprentissage de la programmation ! Ce module 2 va nous montrer les bases de Python qui peut manipuler des nombres et faire de l'arithmétique mais aussi faire du traitement de textes.

Python comme machine à calculer et à écrire

2.1 Tout un programme au menu

2.1.1 Présentation du menu de ce module

MENU DE CE MODULE : MON PREMIER PROGRAMME COMPLET

Note : Voir la vidéo de la section 2.1.1 : Mon premier programme Python

PRÉSENTATION DU MODULE 2

Nous voici arrivés au module 2 de ce cours.

Jusqu'à présent nous avons juste installé le décor.

À partir d'ici, nous allons entamer l'apprentissage du langage Python et de la programmation.

Dans ce module, nous apprendrons comment faire de l'arithmétique et manipuler des textes. Nous verrons aussi comment facilement faire des dessins géométriques.

Dans les modules suivants, nous parlerons des instructions Python (module 3), nous verrons comment structurer un programme en utilisant des fonctions (module 4), et apprendrons à utiliser des structures de données plus compliquées (modules 5 et 6).

Au terme du cours en ligne, vous aurez appris la programmation de base et serez capable d'écrire des programmes élaborés, comme le témoignera le projet que vous aurez réalisé.

Pour chacune des notions que nous voulons vous apprendre, le principe sera un apprentissage en trois phases :

- 1) Nous vous donnerons les bases théoriques pour vous lancer.
- 2) Nous les mettrons en pratique généralement à travers de petits exercices encadrés.
- 3) Vous pourrez ensuite mettre ces connaissances en pratique grâce à de nombreux exercices.

Et n'oubliez pas que pour être à l'aise en programmation, il faut pratiquer intensivement. L'adage « C'est en forgeant que l'on devient forgeron » s'applique parfaitement ici.

2.2 Python comme machine à calculer

2.2.1 Valeurs et expressions

COMMENÇONS PAR L'ARITHMÉTIQUE

Note : Voir la vidéo de la section 2.2.1 : Valeurs et expressions

CONTENU DE LA VIDÉO

La vidéo précédente présente les opérations arithmétiques de base possibles en Python sur des valeurs entières (`int`) ou fractionnaires (`float`). Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```
>>> 3
3
>>> 3 + 5
8
>>> 9 - 5
4
>>> 3 * 4
12
>>> 3 + 4 * 5
23
>>> (3 + 4) * 5
35
>>> 8 - 5 - 2
1
>>> 8 / 4
2.0
>>> 8 / 3
2.6666666666666665
>>> type(3)
<class 'int'>
>>> type(3 + 4 * 5)
<class 'int'>
>>> type(2.66)
<class 'float'>
>>> type(4.0)
<class 'float'>
>>> type(8 / 2)
<class 'float'>
>>> 8 // 4
2
>>> 8 // 3
2
>>> 10 ** 5
100000
>>> 3.14159
3.14159
>>> 314159e-5
3.14159
>>> 0.00001
```

(suite sur la page suivante)

(suite de la page précédente)

```

1e-05
>>> 1.0e-5
1e-05
>>> 2 ** 2 ** 3
256

```

La vidéo en bref

Si nous ouvrons une console Python (dans l'environnement PyCharm), celle-ci attend une instruction de l'utilisateur. On voit les symboles `>>>` qui invitent l'utilisateur à entrer quelque chose.

Dans ce mode, appelé mode interactif, la valeur de ce que l'on évalue est renvoyée. On peut ainsi voir l'effet des différentes opérations arithmétiques et la façon de les écrire : (addition `+`, soustraction `-`, multiplication `*`, division en nombre flottant `/`, division entière `//`, exponentiation `**`), ainsi que les types entiers (`int` : 3, 8, 12, ...) et fractionnaires appelés aussi flottants (`float` : 2.0, 2.6666666666666665, 3.14159, 0.00001, 1.0e-5, ...).

2.2.2 L'arithmétique

TESTER L'ARITHMÉTIQUE

À vous maintenant de « tester les choses » : « tester » dans le jargon informatique signifie exécuter des petits programmes ou des instructions pour voir comment les choses se déroulent (donc ici comment l'interpréteur fonctionne).

Dans PyCharm, ouvrez une console (menu Tools -> Python Console) et expérimentez ce que donnent des calculs utilisant les opérateurs arithmétiques sur les valeurs de type entier (`int`) et fractionnaire (`float`).

Si l'on regarde dans la documentation Python ([The Python Standard Library sur le site python.org](https://docs.python.org/3/library/stdtypes.html)) la liste des opérateurs arithmétiques de base, on obtient (principalement)

+	L'addition
-	La soustraction
*	La multiplication
/	La division réelle (c'est-à-dire dont le résultat est du type float)
//	La division entière tronquée (une explication sur la division est donnée ici)
**	L'exponentiation (appelée également puissance ; une explication sur l'exponentiation est donnée ici)
%	Le modulo (appelé aussi modulus ; pour les nombres entiers positifs, le modulo est défini comme le reste de la division entière ; une explication du modulo est donnée ici).

Nous n'avons pas encore parlé de l'opérateur modulo écrit `%` en Python. Ici nous vous demandons d'être curieux pour comprendre comment fonctionne cet opérateur. La curiosité est un outil essentiel pour bien apprendre à programmer !

De plus Python en mode interactif, par exemple avec une console PyCharm, vous facilite souvent la vie.

Par exemple, si vous encodez `8 % 3` dans une console PyCharm, vous obtiendrez bien 2, qui correspond au reste de la division entière de 8 par 3.

L'opérateur modulo Python fonctionne également avec des nombres négatifs et même avec des nombres fractionnaires.

ASSOCIATIVITÉ ET PRIORITÉ DES OPÉRATEURS

L'associativité et la priorité (appelée également précedence) de ces opérateurs (voir ci-dessous pour une explication), depuis le plus prioritaire vers le moins prioritaire, sont les suivantes :

(expression)	
**	associatifs à droite
* / //	associatifs à gauche
+ -	associatifs à gauche

où les opérateurs sur une même ligne ont le même niveau de priorité.

Priorité

Par exemple, la multiplication, appelée également produit, est plus prioritaire que l'addition : ainsi l'évaluation de $3 + 4 * 5$ vaut 23 (on effectue d'abord la multiplication $4 * 5$ et ensuite l'addition).

Notons que pour $2 ** -1$ le moins est unaire (en fait il faut le voir comme le nombre -1) et donc vaut $2 ** (-1)$ c'est-à-dire 0.5.

Associativité

La plupart des opérateurs sont associatifs à gauche. La soustraction $-$ est associative à gauche : ainsi, $8 - 5 - 2$ est équivalent à $(8 - 5) - 2$.

Par contre, l'exponentiation $**$ est associative à droite : ainsi $2 ** 2 ** 3$ est équivalent à $2 ** (2 ** 3)$, c'est-à-dire 256 et non à $(2 ** 2) ** 3$ qui vaut 64.

Parenthèses

Des parenthèses peuvent être utilisées pour modifier l'ordre d'évaluation dans une expression ; ainsi : $(3 + 4) * 5$ pour réaliser l'addition $(3 + 4)$ avant la multiplication par 5. On évalue donc prioritairement ce qui à l'intérieur des parenthèses.

QUIZ SUR L'ARITHMETIQUE

Dans le quiz suivant, nous vous demandons de nous dire, éventuellement avec l'aide d'une console PyCharm, les résultats de différentes expressions utilisant l'opérateur modulo ou les autres opérateurs arithmétiques.

Note : Voir le quiz de la section 2.2.2

2.2.3 Tester l'arithmétique

PRÉPAREZ UNE MOUSSE AU CHOCOLAT !

Nous avons vu comment faire des calculs avec les différents opérateurs arithmétiques que Python nous fournit.

Avec ces opérateurs en poche, lançons-nous pour faire des calculs utiles !

Par exemple, faisons un peu de cuisine en réalisant la recette de la [mousse au chocolat](#) sur le site [marmiton.org](#) qui est donnée ici :

Ingrédients (pour 4 personnes)

- 3 oeufs
- 100 g chocolat (noir ou au lait)
- 1 sachet de sucre vanillé

Préparation de la recette

- Séparer les blancs des jaunes d'oeufs
- Faire ramollir le chocolat dans une casserole au bain-marie
- Hors du feu, incorporer les jaunes et le sucre
- Battre les blancs en neige ferme et les ajouter délicatement au mélange à l'aide d'une spatule
- Verser dans une terrine ou des verrines et mettre au frais 1 heure ou 2 minimum

QUIZ SUR LA MOUSSE AU CHOCOLAT

La page [mousse au chocolat](#) du site Marmiton donne la quantité de chaque ingrédient quand on adapte le nombre de personnes qui vont manger la recette.

On voit par exemple que pour 7 personnes, 6 oeufs, 175 g de chocolat et 1.75 sachet de sucre vanillé sont requis. Cela correspond aux quantités d'ingrédients pour 4 personnes, divisées par 4 pour calculer les quantités requises par personne, et multipliées par 7 correspondant aux 7 personnes. Si ce calcul pour les oeufs ne tombe pas juste (c'est-à-dire s'il existe une partie fractionnaire), la recette demande un oeuf de plus (c'est ce que l'on appelle la valeur plafond).

Mais si, pour simplifier, nous désirions calculer les ingrédients à l'unité près mais en faisant une simple troncature, c'est-à-dire en ne gardant que la valeur entière, quels résultats aurions-nous ?

Par exemple, les troncatures de 1.3 et de 1.6 donnent tous les deux la valeur 1.

Note : La fonction prédéfinie `int()` peut vous aider. Par exemple `int(1.6)`, qui reçoit comme argument la valeur 1.6 de type fractionnaire (float), donnera la valeur entière 1.

Donc, en utilisant une console PyCharm, pouvez-vous me dire la quantité de chaque ingrédient que je dois avoir pour faire ma recette pour 7 personnes, en tronquant les valeurs calculées à l'unité près ?

Note : Les bons cuisiniers savent que Marmiton a raison dans ses calculs de proportions pour le nombre d'oeufs requis :-)

QUIZ

Note : Voir le quiz de la section 2.2.3

2.3 Python comme machine de traitement de texte

2.3.1 Les expressions chaînes de caractères

LES EXPRESSIONS CHÂÎNES DE CARACTÈRES

La vidéo qui suit présente les notions de base sur les textes appelés chaînes de caractères Python.

VIDÉO SUR LES EXPRESSIONS CHÂÎNES DE CARACTÈRES

Note : Voir la vidéo de la section 2.3.1 : Valeurs et expressions textuelles

CONTENU DE LA VIDÉO

La vidéo précédente présente les chaînes de caractères Python. Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```

>>> "Bonjour"
'Bonjour'
>>> 'Bonjour'
'Bonjour'
>>> 'c'est facile'
File "<input>", line 1
    'c'est facile'
      ^
SyntaxError: invalid syntax
>>> "c'est facile"
"c'est facile"
>>> "Bonjour " + "Michelle"
'Bonjour Michelle'
>>> "bon" * 10
'bonbonbonbonbonbonbonbonbonbon'
>>> 10 * "ha"
'hahahahahahahahahaha'
>>> len("Bonjour")
7

```

La vidéo en bref

"Bonjour" et 'Bonjour' entourés par des doubles ou simples apostrophes, appelés aussi doubles ou simples quotes, sont des chaînes de caractères (textes).

`type('Bonjour')` exprime que le type (la classe) de la valeur 'Bonjour' est `str` pour *string*, terme anglais pour chaîne de caractères, ou plus simplement texte.

Taper dans la console Python `'c'est facile'` renvoie un message d'erreur du type `SyntaxError` puisque l'interpréteur considère que la chaîne de caractères est `'c'` et ne comprend plus la suite.

On peut écrire `"c'est facile"` entourés de double quotes pour obtenir une chaîne de caractères correcte.

Concaténation et répétition

`"Bonjour " + "Michelle"` renvoie `"Bonjour Michelle"`

La concaténation (+) est l'opération qui consiste à coller deux textes ensemble pour n'en former qu'un.

`"bon" * 10` donne `'bonbonbonbonbonbonbonbonbonbon'` et `10 * "ha"` donne `'hahahahahahahahahaha'`

C'est la concaténation de 10 fois le texte `"bon"` ou de 10 fois le texte `"ha"`.

len()

La fonction prédéfinie `len()` donne la longueur de la séquence donnée en argument. Cela signifie que `len("Bonjour")` donne la longueur du texte, c'est-à-dire ici 7 caractères (les doubles quotes étant là pour marquer le début et la fin du texte).

2.3.2 À vous de tester les chaînes de caractères

TESTER LES CHAÎNES DE CARACTÈRES AVEC UNE CONSOLE PYCHARM

À vous de jouer ! Entrez à nouveau dans une console PyCharm et expérimentez comment l'interpréteur Python réalise les manipulations simples de chaînes de caractères.

Pour ce faire, nous vous proposons de regarder ce que donnent les lignes de code suivantes :

```

>>> "bonjour"
>>> 'bonjour'

```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> ' Bonjour '
```

```
>>> "C'est facile"
```

Il est également possible d'appliquer des fonctions à des chaînes de caractères. Par exemple, la fonction prédéfinie `len()` donne la longueur de la séquence passée en argument. Cela signifie que `len("bonjour")` donne la longueur du texte soit 7 caractères (les doubles quotes étant là pour marquer le début et la fin du texte).

Afin de voir si vous avez compris l'utilité de la fonction `len()`, nous vous proposons le quiz suivant.

QUIZ SUR LA FONCTION LEN

Note : Voir le quiz de la section 2.3.2

2.4 Les variables pour changer

2.4.1 Les variables

LES VARIABLES POUR RETENIR LES TRAITEMENTS

La vidéo qui suit présente les notions fondamentales de variables et d'*assignment*, appelée également *affectation*, en Python.

VIDÉO SUR LES VARIABLES

Note : Voir la vidéo de la section 2.4.1 : Variable et assignation

CONTENU DE LA VIDÉO

La vidéo précédente présente les variables Python. Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```
>>> x = 3
```

```
>>> x
```

```
3
```

```
>>> y = 4 * 3
```

```
>>> y
```

```
12
```

```
>>> y = 4
```

```
>>> y
```

```
4
```

```
>>> x + 5
```

```
8
```

```
>>> z = x + 5
```

```
>>> y = y + 1
```

```
>>> y
```

```
5
```

```
>>> r = 8 / 3
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> r
2.6666666666666665
>>> type(y)
<class 'int'>
>>> type(r)
<class 'float'>
>>> mon_message = 'bonjour'
>>> x, y, z
(3, 5, 8)
>>> ma_variable = x
>>> ma_variable
3

```

La vidéo en bref

Les variables permettent de retenir des valeurs; l'instruction d'*affectation*, appelée également *assignation*, donne une valeur à une variable.

Une assignation a la forme suivante

```
nom = valeur
```

où **à gauche** du symbole d'assignation =, *nom* est le nom de la variable, choisi par le programmeur, et **à droite** du symbole = est donnée la valeur à assigner. Par exemple :

```

x = 3
y = 4 * 3
z = x + 5
y = y + 1
r = 8 / 3
mon_message = 'bonjour'

```

Après l'assignation, (par exemple $x = 3$), la variable assignée a une valeur et un type (ici x vaut la valeur *entière* 3);

$x + 5$ utilise la valeur de la variable x pour faire les calculs (ici comme x vaut 3 : $3 + 5$ c'est-à-dire 8);

`type(x)` renseigne que la variable x , donnée en argument de la fonction prédéfinie `type()` est de type entier (son contenu est un entier). De façon raccourcie on dit que x est un entier.

Certains noms sont des mots réservés (mots-clés) Python : ainsi `if = 0` ne fonctionne pas : l'interpréteur donne une erreur de syntaxe car `if` est un mot-clé Python.

Notons plus précisément qu'une valeur en Python sera un *objet* et une variable le *nom* d'un objet.

2.4.2 Code avec variables

À VOUS DE JOUER !

L'exercice suivant vous demande d'écrire votre premier code avec des variables.

Pour cela, reprenez la préparation de la mousse au chocolat donnée dans l'activité 2.2.3.

Nous vous demandons de refaire les calculs dans une console PyCharm mais cette fois, en assignant à une variable (de nom) n le nombre de personnes pour lesquelles il faut préparer de la mousse au chocolat, et qui assignent dans les variables respectives `oeufs`, `chocolat` et `sucre_vanille`, une approximation (en tronquant les valeurs à l'unité près) de la quantité d'ingrédients (en grammes ou unités) à avoir pour la préparation. Pour la préparation, il faudra au moins mettre un sachet de sucre vanillé.

Note : En plus de la fonction prédéfinie `int(x)` vue précédemment, utilisez la fonction prédéfinie `max(x, y)` qui renvoie le

maximum entre deux ou plusieurs valeurs. Par exemple `max(x, 1)` vaut 1 si `x` est inférieur à 1, `x` sinon. Terminez l'exécution en vérifiant la valeur des variables :

```
n, oeufs, chocolat, sucre_vanille
```

Pour `n` valant 7, les valeurs pour `oeufs`, `chocolat` (en grammes) et `sucre_vanille` sont respectivement de 5, 175 et 1.

On voit que l'assignation en Python revient réellement à donner un nom à une valeur.

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution : nous vous en proposons une.

```
n = 12
oeufs = int(3 * n / 4)
chocolat = int(100 * n / 4)
sucre_vanille = max(int(n / 4), 1)
```

RÈGLE DE BONNE PRATIQUE

À toute fin utile, voici une règle de bonne pratique qui vous permettra de rendre votre code plus lisible et souvent plus efficace : on utilise généralement des noms de variables qui ont un sens en fonction de leur contenu. Par exemple `sucre_vanille`. En Python, les noms des variables ne peuvent contenir d'accents ni de caractères non alphanumériques (un caractère alphanumérique signifie une lettre ou un chiffre) excepté le caractère souligné « `_` », et ne peuvent pas commencer par un chiffre. Par exemple `sucre_vanille`, `s9` ou `oeufs` sont des noms corrects pour des variables mais par exemple `sucre_vanillé` (qui contient un accent) ou `9s` (qui commence par un chiffre) ne sont pas des noms de variables valides.

Note : PyCharm analyse nos scripts pour déterminer s'ils respectent bien les « règles de bonnes pratiques » (nous en reparlerons plus loin) : si les mots utilisés par exemple pour donner un nom aux variables ne sont pas dans ses dictionnaires, PyCharm nous avertit qu'il y a probablement un typo (mais peut-être que vous l'avez délibérément orthographié ainsi). C'est pour cette raison que nous avons ajouté un dictionnaire français contenant les mots avec et sans accents ce qui permet de donner dans nos codes des noms de variables sans les accents (par exemple `cafe = 5` si notre programme parle de café).

UN PEU DE VOCABULAIRE : VARIABLE ET CONSTANTE

Dans un programme, on distingue les valeurs qui peuvent changer, comme la valeur des variables `n`, `oeufs`..., des valeurs qui restent inchangées tout au long du programme, et que l'on appelle *constantes* dans le jargon informatique. Comme son nom l'indique la valeur d'une constante n'est jamais modifiée lors de l'exécution du programme. Ainsi, de façon évidente, les valeurs 0, 1 ou 7 sont des constantes. Plus loin, on parlera de la valeur `pi` comme étant une constante.

SYNTAXE ET SÉMANTIQUE

Rappelons la définition de ces deux mots courants en informatique et dans notre cours en ligne.

Syntaxe : pour un programme Python, est l'ensemble des règles grammaticales que l'on doit suivre pour que le programme soit a priori compréhensible par l'interpréteur. Parmi les règles de syntaxe à suivre, nous avons vu que l'assignation doit avoir un nom de variable à gauche du signe « `=` » et une valeur ou une expression à droite.

Sémantique : exprime le « sens », la « signification ». Parmi les erreurs sémantiques dans un code Python, on a par exemple le fait d'utiliser un nom (identificateur) non défini.

Par exemple `x = z` est une instruction Python syntaxiquement correcte. Mais, si `z` n'a encore reçu aucune valeur, par exemple, via l'assignation (`z` n'a pas encore été définie), l'interpréteur Python qui exécute cette instruction produira une erreur (`NameError`) : c'est une erreur sémantique.

Note : Voir le quiz de la section 2.4.3

2.5 PyCharm en mode script, entrées et sorties

2.5.1 Manipuler des scripts avec PyCharm

DES SCRIPTS QUI COMMUNIQUENT : C'EST MIEUX !

La capsule suivante introduit les deux fonctions `input` et `print` qui seront bien utiles pour que le code Python puisse communiquer avec son utilisateur. La notion de script Python est aussi introduite.

VIDÉO SUR LES SCRIPTS

Note : Voir la vidéo de la section 2.5.1 : `print` et `input`

CONTENU DE LA VIDÉO

La vidéo précédente présente les scripts Python ainsi que les fonctions `input` et `print`. Son résumé est groupé avec celui de la vidéo de l'onglet 3 de cette section, un peu plus loin dans ce cours ; cette seconde vidéo complète l'explication. Le résumé des deux vidéos est disponible dans cet onglet 3.

2.5.2 Python Tutor et les diagrammes d'état

UTILISATION DE PYTHON TUTOR

Nous avons déjà présenté Python Tutor dans le module précédent. Python Tutor est un outil en ligne permettant d'exécuter pas à pas des petits scripts Python en visualisant l'effet de chaque instruction Python.

Avant de montrer de façon plus précise comment fonctionne Python Tutor, commençons par expliquer pourquoi diable nous introduisons un outil de plus dans ce cours en ligne. La réponse est double :

- d'une part Python Tutor permet d'interpréter le code Python en s'arrêtant à chaque instruction pour voir son effet,
- d'autre part, après chaque instruction, Python Tutor représente le « diagramme d'état » illustrant l'état du programme à ce moment-là.

Comme nous allons le voir, un diagramme d'état est un schéma qui montre graphiquement comment les variables et valeurs manipulées dans un script Python sont stockées en mémoire. Si au départ cette information ne vous sera pas d'une grande utilité, petit à petit nous verrons qu'elle est essentielle pour comprendre comment nos codes fonctionnent.

Python Tutor n'est donc pas un simple interpréteur ; c'est un outil d'apprentissage de Python (et d'autres langages) principalement pour les débutants. Python Tutor nous sera d'une grande utilité, en particulier pour visualiser les diagrammes d'états montrant l'état de codes fournis, à chaque étape de l'exécution, ce qu'un interpréteur « normal » comme PyCharm ne fait pas. Mais Python Tutor n'est pas un interpréteur complet de scripts en ligne comme ce qui vous est fourni via PyCharm. Ainsi, Python Tutor ne permet l'exécution que des scripts utilisant un petit sous-ensemble d'instructions Python.

Pour bien expliquer les nouveaux concepts Python, nous incluons fréquemment des exemples Python Tutor intégrés dans ce cours en ligne. Mais libre à vous de mettre dans une fenêtre Python Tutor un code Python de votre choix pour visualiser son exécution étape par étape et bien comprendre son fonctionnement grâce aux diagrammes d'état fournis par Python Tutor.

Pour vous aider à utiliser Python Tutor, la petite vidéo qui suit fait une démonstration de son utilisation. Libre à vous de la visionner maintenant ou quand vous en aurez besoin.

RÉSUMÉ DE LA MANIÈRE D'UTILISER PYTHON TUTOR

- Sur votre navigateur Web ouvrez la page `pythontutor.com`.
- Cliquez sur “Start visualizing your code now” qui vous permet de visualiser comment s’exécute, pas à pas, le script que vous lui donnez dans la zone centrale de la fenêtre.
- N’oubliez pas de mettre l’option python 3.6 ou ultérieur et de sélectionner les bonnes options proposées sous cette fenêtre. Nous vous conseillons de prendre les options par défaut, et si vous désirez que les diagrammes d’état fournissent plus de détails sur les valeurs, remplacez l’option « inline primitives & nested objects » par « render all objects on the heap ».
- Vous pouvez ensuite exécuter pas à pas en avant ou en arrière le script en cliquant sur le bouton « Visualize Execution » et ensuite utiliser les boutons « Forward », « Back », « First », « Last » ou encore le curseur.
- L’affichage résultant des `print` est donné dans une sous-fenêtre « Print output », et lors des `input` une fenêtre sous l’intitulé « Enter user input below : » s’ouvre vous permettant d’entrer la donnée après avoir cliqué sur le bouton « submit ».
- Pour revenir en mode édition, cliquez sur le bouton « Edit the code ».

Dans les exemples Python Tutor intégrés au code, toute la partie initialisation a déjà été réalisée pour vous. Il ne vous reste plus qu’à cliquer sur les boutons pour faire avancer ou reculer l’exécution ainsi qu’à introduire les `inputs` demandés.

La vidéo qui suit illustre ce qui vient d’être expliqué.

PYTHON TUTOR

Note : Voir la vidéo de la section 2.5.2 : Python Tutor

À VOUS DE JOUER !

Dans le but d’apprendre à manipuler Python Tutor, reprenez votre script rédigé à la section précédente qui calcule les quantités d’ingrédients pour la préparation de la mousse au chocolat. Modifiez-le dans Python Tutor pour lire la valeur de `n` (`input` de `n`) et écrire les résultats grâce à des `print` : la valeur des variables `oeufs`, `chocolat` et `sucres_vanille`.

Ensuite, visualisez comment celui-ci s’exécute pas à pas dans une fenêtre de Python Tutor.

ANIMATIONS PYTHON TUTOR INTÉGRÉES

Pour rappel, dans ce cours, nous utiliserons également Python Tutor de façon intégrée, c’est-à-dire au sein même des pages du cours. Par exemple, voici deux animations Python Tutor intégrées de l’exemple précédent sur la mousse au chocolat avec `input` et `print`. La première utilise l’option « render all objects on the heap », la seconde l’option « inline primitives & nested objects ».

Note : Voir exemples Python Tutor dans le cours en ligne

2.5.3 Commentons notre programme

DES SCRIPTS COMMENTÉS : C’EST ENCORE MIEUX !

Note : Voir la vidéo de la section 2.5.3 : Commentaires et docstrings

CONTENU DES VIDÉOS

Les deux vidéos sur les scripts Python ainsi que les fonctions `input` et `print`, présentées dans l'onglet précédent et l'onglet courant de la présente section, sont résumées ici.

Nous donnons ici les scripts qui y sont utilisés pour la présentation suivis d'un résumé des explications données s'y rapportant.

Scripts utilisés dans les vidéos

```
rayon = 5.0
circ = 2 * 3.14 * rayon # l'astérisque manque
aire = 3.14 * rayon ** 2
circ, aire
```

```
rayon = 5.0
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon ** 2
circ, aire
```

```
rayon = 5.0
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon ** 2
circ, aire
print(circ, aire)
```

```
rayon = float(input())
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon ** 2
print(circ, aire)
```

```
rayon = float(input("Veuillez donner le rayon : "))
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon ** 2
print("Circonférence :", circ)
print("Aire          :", aire)
```

```
""" Auteur: Sébastien Hoarau
    date : juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque
            l'aire du disque
    """

rayon = float(input("Veuillez donner le rayon : ")) # lecture du rayon de mon disque
circ = 2 * 3.14 * rayon                             # calcul de la circonférence
aire = 3.14 * rayon ** 2                             # calcul de l'aire
# Affichage des résultats
print("Circonférence : ", circ)
print("Aire          : ", aire)
```

La vidéo en bref

Les deux fonctions `input` et `print` sont introduites. Elles sont bien utiles pour que le code Python puisse communiquer avec

son utilisateur. La notion de script Python est aussi introduite.

- 1) Le premier script est édité et ensuite sauvé par exemple dans le fichier `disque.py`. Son exécution par l'interpréteur Python donne une erreur de syntaxe : l'interpréteur lorsqu'il désire calculer la valeur de `circ` ne comprend pas ce qui est demandé. En effet cette ligne de code n'est pas correcte : il manque l'opérateur de multiplication `*`.
- 2) Après avoir corrigé mon script en ajoutant le symbole de multiplication, je redemande, avec la commande **Run**, à l'interpréteur d'exécuter le code du script `disque.py` et cette fois il se termine avec un `exit code 0` qui signifie que l'exécution s'est bien déroulée. Mais nous constatons que le script n'a pas communiqué de résultats !
- 3) Le troisième script permet avec la fonction `print` d'*afficher*, on dit aussi *imprimer*, les résultats donnés en arguments du `print`.
- 4) Le quatrième script permet de recevoir grâce à la fonction `input()` une *donnée* que l'utilisateur peut encoder, donnée qui est traduite en valeur fractionnaire grâce à la fonction prédéfinie `float()`.
- 5) Le cinquième script ci-dessus demande l'entrée de données (`input()`) avec un texte explicatif en argument, l'affichage de résultats (`print()`) également avec un texte explicatif à chaque `print`.
- 6) Le dernier script contient des *commentaires*. De façon générale, pour rendre un script plus lisible pour les autres programmeurs qui voudraient comprendre son fonctionnement, on peut rajouter des explications. Pour cela on peut, à la fin de chaque ligne de code, mettre le caractère `#` (croisillon) suivi de texte explicatif. Ce texte s'appelle du *commentaire* et ne sera pas utilisé par l'interpréteur. Je peux aussi, n'importe où, rajouter un commentaire multiligne entouré de 3 simples quotes ou double quotes. Je peux mettre une explication plus complète de ce que fait le script dans un *docstring* initial :
 - l'auteur
 - la date
 - le but du programme
 - les données reçues (`input`)
 - les résultats affichés (`print`)

les caractères croisillon et dièse

Le caractère `#` utilisé en Python pour marquer les commentaires est le caractère *croisillon* ; celui-ci a deux barres obliques parallèles coupées par deux barres horizontales. Ce caractère est souvent erronément appelé dièse (`#`). Ce dernier, utilisé par exemple dans les partitions musicales, a deux barres verticales coupées par deux barres obliques parallèles.

CODE COMPLET COLORISÉ POUR L'EXEMPLE « MOUSSE AU CHOCOLAT »

Nous pouvons maintenant visualiser un script complet commenté. Pour illustrer cela, donnons deux scripts complets solutionnant le problème de la mousse au chocolat :

- une version avec assignation des résultats dans des variables
- et une version améliorée où les calculs se font directement lors des `print`.

Solution avec variables

```
"""Auteur: Thierry Massart
   Date : 5 décembre 2017
   But du programme : calcule les ingrédients nécessaires
   pour préparer de la mousse au chocolat pour n personnes
   Entrée: n (nombre de personnes)
   Sorties: nombre d'oeufs,
            quantité en gramme de chocolat,
            nombre de sachets de sucre vanillé
   """
n = int(input("nombre de personnes : "))      # entrée: nombre de personnes
oeufs = int(3*n/4)                            # nombre d'oeufs nécessaires
chocolat = int(100*n/4)                      # quantité de chocolat nécessaire
sucre_vanille = max(int(n/4), 1)             # quantité de sucre nécessaire
```

(suite sur la page suivante)

(suite de la page précédente)

```
print("nombre d'oeufs : ", oeufs)           # affiche les résultats
print("quantité de chocolat (g) : ", chocolat)
print("quantité de sucre_vanillé : ", sucre_vanille)
```

Solution améliorée

```
"""Auteur: Thierry Massart
Date : 5 décembre 2017
But du programme : calcule les ingrédients nécessaires
pour préparer de la mousse au chocolat pour n personnes
Entrée: n (nombre de personnes)
Sorties: nombre d'oeufs,
         quantité en gramme de chocolat,
         nombre de sachets de sucre vanillé
"""
n = int(input("nombre de personnes : ")) # entrée: nombre de personnes
print("nombre d'oeufs : ", int(3*n/4)) # calcule et affiche les résultats
print("quantité de chocolat en grammes : ", int(100*n/4))
print("quantité de sucre_vanillé : ", max(int(n/4), 1))
```

2.5.4 Réalisation des exercices UpyLaB du module

MISE EN PRATIQUE AVEC UPYLAB

Nous voici à la dernière étape d'apprentissage pour la matière de ce module : l'apprentissage autonome. Pour cela, nous vous demandons de réaliser les exercices 2.1 à 2.3 du Module 2 qui vous sont proposés dans le cadre de cette activité.

Note : En cas de problème d'affichage, essayez de recharger la page. Il se peut que votre navigateur ne sache pas bien afficher le code UpyLaB, et que vous deviez en utiliser un autre.

DÉVELOPPEMENT, MISE AU POINT ET DÉBOGAGE D'UN CODE

Faire les exercices UpyLaB va vous demander de *développer du code Python* résolvant les problèmes demandés. En général, au début, votre code ne résoudra pas (exactement) le problème demandé : nous dirons simplement que votre code est faux ! Dans ce cas, vous devrez le *déboguer*.

Déboguer son code (debug en anglais) correspond à y faire la chasse aux erreurs. La pratique de la programmation apprend comment déboguer efficacement un programme. Nous essayerons tout au long de ce cours de vous donner certains conseils pour y arriver. En voici quelques uns :

CONSEILS ET CONSIGNES LORS DE LA RÉALISATION D'EXERCICES UPYLAB

- 1) Lors du développement de chacun de vos codes, il est plus que vivement conseillé d'utiliser un environnement de développement comme **PyCharm** ou **Python Tutor** pour *tester* ou même *déboguer* pas par pas votre code avant de passer aux tests *UpyLaB*. En effet, faire tester directement par UpyLaB un code dont vous n'avez pas une certaine confiance dans le fait qu'il soit correct s'avère souvent être un gouffre au niveau du temps. En effet, pour valider votre code, UpyLaB ajoute du code supplémentaire qu'il va ensuite exécuter : si votre code est erroné à une ligne donnée, il est possible qu'UpyLaB évoque un autre numéro de ligne pour cette erreur ou renvoie un message peu explicite. Pour cette raison, il est recommandé de ne pas utiliser UpyLaB lors des développements de vos codes, mais uniquement pour valider s'ils sont bien corrects.

La démarche la plus efficace pour développer du code en Python est donc :

- de comprendre exactement ce qui vous est demandé,
- de développer une solution dans PyCharm en la testant,

- de tester l'ensemble de votre code dans UpyLaB (en le transférant dans UpyLaB grâce au copier / coller du code dans PyCharm) pour faire valider votre script.
- 2) UpyLaB est parfois « psychorigide » dans le sens où, si votre code ne fait pas **exactement** ce qu'il demande, UpyLaB le considère comme faux. En particulier dans les consignes, votre code à tester par UpyLaB :
 - ne doit pas avoir d'arguments dans les appels à `input`.
 - ne doit pas imprimer autre chose que le résultat dans les `print`.
 Ainsi, il **faut** écrire :

```
a = float(input())
print(a)
```

plutôt que :

```
a = float(input("a = "))
print('a vaut :', a)
```

En effet UpyLaB teste votre programme en l'exécutant plusieurs fois. Durant ces tests, la sortie (`print`) du code proposé sur différentes entrées fournies automatiquement est comparée à la sortie attendue, produite par une solution de référence. Si votre code fait des sorties supplémentaires avec les fonctions `input` ou `print`, il sera jugé faux par UpyLaB lors des validations ! Vous expérimenterez sûrement cela tôt ou tard. Cela vous obligera à une rigueur à laquelle les novices en programmation sont généralement peu habitués.

Attention : Le but des exercices UpyLaB est de vous rendre **autonome** en programmation. C'est vous qui devez trouver la ou une solution, en terme de code Python, à chaque exercice demandé. Si vous ne voyez pas comment y arriver ou qu'UpyLaB ne valide toujours pas votre code, voici quelques conseils :

1. Relire l'énoncé et les consignes associés à l'exercice ainsi que les éventuels conseils.
2. Lire la FAQ générale à propos des exercices UpyLaB.
3. Lire, si elle existe, la FAQ spécifique à l'exercice réalisé.
4. Si rien de tout cela ne fonctionne, poser votre question sur le Forum dédié aux exercices, en étant le plus précis possible, *sans mettre votre code complet ni substantiel* mais en mettant des copies d'écran des résultats fournis par UpyLaB.

2.5.5 Exercice UpyLaB 2.1 - Parcours Vert, Bleu et Rouge

Le but de cet exercice est de vérifier que vous savez définir des variables, et leur affecter des valeurs des différents types rencontrés dans le cours.

Écrire un programme qui assigne :

- la valeur entière 36 à la variable `x` ;
- la valeur entière résultat de 36 fois 36 à la variable `produit` ;
- la valeur entière résultat de la division entière de 36 par 5 à la variable `div_entiere` ;
- la valeur entière résultat de 15 exposant 15 à la variable `expo` ;
- la valeur float 3.14159 à la variable `pi` ;
- la valeur chaîne de caractères "Bonjour" à la variable `mon_texte`.

Consignes

Attention, dans cet exercice, il n'y a rien à afficher, donc vous ne ferez aucun appel à la fonction `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter en ajoutant des instructions lui permettant de vérifier que les variables attendues existent et sont bien affectées des valeurs attendues.

Si vous souhaitez tester votre code **dans PyCharm**, pensez à ajouter les instructions :

```
print("x =", x)
print("produit =", produit)
print("div_entiere =", div_entiere)
...
```

etc, pour visualiser le contenu de vos variables.

- Veillez à bien respecter les noms de variables attendus.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.1*.

2.5.6 Exercice UpyLaB 2.2 - Parcours Vert, Bleu et Rouge

Le but de cet exercice est de vérifier que vous savez lire des données en entrée et les affecter à des variables.

Écrire un programme qui imprime la moyenne arithmétique de deux nombres de type float lus en entrée.

Moyenne arithmétique de deux nombres a et b : $\frac{a+b}{2}$

Exemple 1

Avec les données lues suivantes :

```
2.0
3.0
```

le résultat à imprimer vaudra :

```
2.5
```

Exemple 2

Avec les données lues suivantes :

```
4.2
3.8
```

le résultat à imprimer vaudra :

```
4.0
```

Consignes

Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.

En particulier, il ne faut rien écrire à l'intérieur des appels à `input(float(input()))` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Portez attention aux règles de priorité des opérateurs arithmétiques.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.2*.

2.5.7 Exercice UpyLaB 2.3 - Parcours Vert, Bleu et Rouge

Le but de cet exercice est de vous familiariser avec la lecture (`input()`) de données et l'impression (`print()`) de résultats.

La **règle de trois** est une méthode pour trouver le quatrième terme parmi quatre termes ayant un même rapport de proportion $\frac{a}{b} = \frac{c}{d}$ lorsque trois de ces termes sont connus.

Elle utilise le fait que le produit des premier et quatrième termes est égal au produit du second et du troisième : $a.d = b.c$ et donc $d = \frac{b.c}{a}$

Exemple : si chacun mange autant de chocolat et que pour 4 personnes il en faut 100 grammes, pour 7 personnes il en faudra donc d tel que $\frac{4}{100} = \frac{7}{d}$

D'où $d = \frac{7.100}{4}$ grammes = 175 grammes.

Écrire un programme qui lit des valeurs de type float pour a , b et c et qui affiche la valeur de d correspondant à la règle de trois.

Exemple 1

Avec les données lues suivantes :

```
4.0
100.0
7.0
```

le résultat à imprimer vaudra :

```
175.0
```

Exemple 2

Avec les données lues suivantes :

```
3.5
0.5
8.0
```

le résultat à imprimer vaudra :

```
1.1428571428571428
```

Remarque : Du fait du manque de précision dans les calculs avec les nombres de type float, votre résultat pourra légèrement différer de celui indiqué ci-dessus. Ce n'est pas un problème, car UpyLaB acceptera toute réponse suffisamment proche du résultat attendu, avec une tolérance d'environ $1.0e-5$.

Consignes

Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.

En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.3*.

2.6 Quelques fonctions prédéfinies, les modules math et turtle

2.6.1 Exemples d'utilisations de fonctions prédéfinies, et des modules math et turtle

PRÉSENTATION

Les capsules suivantes montrent que Python fournit au programmeur des modules avec des constantes et des fonctions prédéfinies, c'est-à-dire déjà écrites, qu'il peut utiliser à sa guise soit directement, soit en les important.

VIDÉO SUR LE MODULE MATH

Note : Voir la vidéo de la section 2.6.1 : Le module math

VIDÉO SUR LE MODULE TURTLE

Note : Voir la vidéo de la section 2.6.1 : Le module turtle

CONTENU DES VIDÉOS

Les deux vidéos précédentes ont introduit les modules Python `math` et `turtle`.

Nous donnons ici les scripts qui y sont utilisés pour la présentation suivis d'un résumé des explications données s'y rapportant.

Scripts et consoles utilisés dans les vidéos

Le module math

```

""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = 3.14 * 2 * rayon # calcul de la circonférence
aire = 3.14 * rayon ** 2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)

```

```

""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

pi = 3.14159

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = pi * 2 * rayon # calcul de la circonférence
aire = pi * rayon ** 2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)

```

```

""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

import math

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = math.pi * 2 * rayon # calcul de la circonférence
aire = math.pi * rayon ** 2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)

```

(suite sur la page suivante)

(suite de la page précédente)

```
print("Aire          :", aire)
```

```
>>> import math
>>> help(math)
...
>>> dir(math)
['__doc__', '__file__', '__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',
→ 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
→ 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
→ 'tanh', 'tau', 'trunc']
>>> math.cos(0.5)
0.8775825618903728
>>> math.cos(math.pi)
-1.0
>>> from math import cos, pi
>>> cos(pi)
-1.0
```

```
""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
    """

from math import pi

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = pi * 2 * rayon # calcul de la circonférence
aire = pi * rayon ** 2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)
```

Le module turtle

```
import turtle
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```

```
import turtle

turtle.up()
turtle.goto(-150,-150)
turtle.down()
```

(suite sur la page suivante)

(suite de la page précédente)

```

turtle.color("blue")
turtle.begin_fill()
turtle.goto(150,-150)
turtle.goto(150,150)
turtle.goto(-150,150)
turtle.goto(-150,-150)
turtle.end_fill()
turtle.done() # turtle.mainloop()

```

La vidéo en bref

Cette section montre que Python fournit au programmeur des modules avec des constantes et des fonctions prédéfinies, c'est-à-dire déjà écrites, qu'il peut utiliser à sa guise soit directement, soit en les important.

Le module math

Au début du script, après le *docstring initial*, on peut rajouter la ligne de code

```
import math
```

Dans ce cas, pour utiliser pi, je dois spécifier que c'est l'attribut du module math en tapant `math.pi` qui me donne la valeur approximative de pi (avec les 15 ou 16 premiers chiffres corrects vu la précision possible).

Si je n'ai besoin que de pi et cos du module math par exemple, je peux préciser et écrire

```
from math import cos, pi
```

Et alors `cos(pi)`, qui appelle la fonction `cos()` avec pi en argument, et qui donc demande la valeur du cosinus de pi, est bien compris par l'interpréteur Python.

Le module turtle

Le module `turtle` permet de dessiner des figures de façon très simple. `turtle` peut être vue comme une tortue qui porte une plume. Quand elle se déplace, soit la plume est descendue, ce qui est le cas au début, et dans ce cas elle trace une ligne lors de ses déplacements, soit la plume est relevée et dans ce cas, elle ne trace rien.

Une liste de commandes `turtle`, les plus communes dont nous allons nous servir avec leur explication, est donnée plus loin dans le cours.

Pour laisser la fenêtre ouverte à la fin du programme jusqu'à ce que l'utilisateur décide de la fermer en cliquant sur le bouton rouge de fermeture de la fenêtre où `turtle` a dessiné, il suffit d'ajouter comme dernière instruction du code : `turtle.done()` ou bien `turtle.mainloop()`.

2.6.2 Quelques fonctions prédéfinies et fonctions turtle très utilisées

FONCTIONS PRÉDÉFINIES

Dans la vidéo précédente, nous avons déjà utilisé certaines fonctions prédéfinies; en voici une liste non exhaustive qui vous sera sûrement bien utile pour résoudre des exercices dans le cadre de ce cours ou de projets futurs :

- `abs(x)`
- `dir(x)`
- `divmod(x, y)`
- `float(x)`
- `help(x)`
- `input()`
- `int(x)`
- `max(a, b, ...)`
- `min(a, b, ...)`

```
— print()
— round(x, y)
— sum(a, b, ...)
— type(x)
```

N'hésitez pas à vous les approprier en les manipulant avec PyCharm avec différents arguments donnés à la fonction appelée, à utiliser la fonction `help()` ou même à aller voir dans la documentation python3 pour bien comprendre ce que ces fonctions font.

Par exemple pour comprendre comment fonctionne la fonction `round()`, tapez dans une console PyCharm `help(round)` (et si l'anglais n'est pas votre fort, n'oubliez pas que sur internet des traducteurs automatiques en ligne peuvent vous aider).

Par exemple dans une console PyCharm (session interactive), tapez

```
>>> help(divmod)
```

pour savoir l'effet de cette fonction. L'explication utilise le mot invariant : cela signifie un fait toujours vrai.

isinstance

Si l'on veut tester si une valeur `ma_variable` est de type `int` par exemple, la fonction prédéfinie `isinstance` peut être utilisée avec la fonction booléenne `isinstance(ma_variable, int)` plutôt que `type(ma_variable) is int`. La différence entre les deux façons de faire n'est visible qu'en programmation orientée-objet (si dans une console python vous tapez `help(isinstance)`, il est dit que `isinstance(obj, class)` teste si `obj` est une instance d'une classe ou une sous-classe de `class`). Dans le cadre de ce cours, comme la notion de sous-classe d'une classe n'a pas été vue et n'est pas utilisée, les deux utilisations sont équivalentes.

Aide-mémoire : Pour vous aider tout au long de ce cours, nous avons confectionné un aide-mémoire, qui explique succinctement l'effet des fonctions et méthodes prédéfinies les plus couramment utilisées.

L'aide-mémoire est disponible à l'adresse <https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@aide-memoire-Python.pdf>

Nous vous conseillons même de l'imprimer, et de l'avoir sous la main quand vous programmez. Notez que jusqu'à présent nous n'avons étudié qu'une petite partie du contenu de cet aide-mémoire. Soyez patient, la suite arrive !

Module math

Le module `math` est particulièrement utile pour écrire des scripts résolvant des problèmes mathématiques. Le but de ce cours n'est sûrement pas de connaître le contenu de modules comme `math` par coeur. Par contre, il est important lorsque nous écrivons un programme Python de savoir trouver de l'information complémentaire. Une première exploration du contenu d'un module, comme le module `math`, consiste à ouvrir une console PyCharm et à exécuter :

```
>>> help('math')
```

Soyez curieux et explorez ce module ou le module `turtle` par exemple !

Module turtle

Comme annoncé dans la vidéo précédente, dans ce cours, nous allons également utiliser le module `turtle` qui va nous permettre de dessiner des figures de façon très simple sur une fenêtre graphique.

Comme les exemples dans la vidéo le montrent, le module `turtle` utilise deux façons pour se déplacer : soit en demandant avec les verbes `forward`, `backward`, `left`, `right` d'avancer (`forward`), de reculer (`backward`), de tourner à gauche (`left`) ou à droite (`right`), soit en spécifiant, avec des `goto`, les coordonnées (`x`, `y`) dans le plan où la tortue doit se déplacer.

- la valeur de l'*abscisse* `x` donnant la position gauche / droite (plus la valeur est négative, plus elle est à gauche de la fenêtre, plus elle est positive, plus elle est à droite) ;
- la valeur de l'*ordonnée* `y` donnant la position bas / haut (plus la valeur est négative, plus elle est en bas de la fenêtre, plus elle est positive, plus elle est en haut).

Initialement, la tortue se trouve au centre de la fenêtre, en coordonnée `(0, 0)`. Utiliser dans des `goto` des valeurs trop petites (par exemple `-400`) ou trop grandes (par exemple `400`) fait sortir la tortue de la fenêtre.

Voici une liste de commandes `turtle`, les plus communes, dont nous allons nous servir :

Commande	Effet
<code>reset()</code>	On efface tout et on recommence
<code>goto(x, y)</code>	Aller à l'endroit de coordonnées x, y
<code>forward(distance)</code>	Avancer d'une distance donnée
<code>backward(distance)</code>	Reculer
<code>up()</code>	Relever le crayon (pour pouvoir avancer sans dessiner)
<code>down()</code>	Abaissier le crayon (pour recommencer à dessiner)
<code>color(couleur)</code>	couleur peut être 'red', 'blue', etc.
<code>left(angle)</code>	Tourner à gauche d'un angle donné (exprimé en degrés)
<code>right(angle)</code>	Tourner à droite
<code>width(épaisseur)</code>	Choisir l'épaisseur du tracé
<code>begin_fill()</code>	Début de la zone fermée à colorier
<code>end_fill()</code>	Fin de la zone fermée à colorier
<code>write(texte)</code>	texte doit être une chaîne de caractères
<code>done()</code> ou <code>mainloop()</code>	Attend que l'utilisateur ferme la fenêtre

Nous vous invitons à ouvrir une console PyCharm et à essayer ces commandes.

TESTONS LES FONCTIONS TURTLE

Pour bien visualiser le rôle des fonctions couramment utilisées par `turtle`, créez un script PyCharm qui contient le code ci-dessous et exécutez-le (Run). Afin de vous aider à analyser l'effet de chaque instruction, nous vous suggérons de commenter chaque ligne en expliquant ce qu'elle fait et en ayant découvert quelle figure est dessinée par ce code.

Note : Attention, surtout n'appellez pas votre script `turtle.py` qui est le nom utilisé pour le module `turtle` lui même. Si vous faites cela, l'import du module `turtle` ne sera pas réalisé.

```
import turtle
turtle.up()
turtle.shape('turtle')
turtle.goto(-80,0)
turtle.color('blue')
turtle.down()
turtle.begin_fill()
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.end_fill()
turtle.hideturtle()
turtle.done()
```

PROPOSITION DE SOLUTION

```
""" auteur: Thierry Massart
    date: 7 décembre 2017
```

(suite sur la page suivante)

(suite de la page précédente)

```

Trace avec turtle une étoile dont les extrémités sont bleues
"""
import turtle                # importation du module turtle
turtle.up()                  # tant que la tortue est en mode "up",
                             # son déplacement ne trace rien
turtle.shape('turtle')      # change la forme de la tortue (en tortue)
turtle.goto(-80,0)           # la tortue se place en coordonnées (-80, 0)
                             # (-80 pour l'axe des "x" et 0 l'axe des "y")
turtle.color('blue')         # la tortue est bleue
turtle.down()                # tant que la tortue est "down",
                             # elle tracera la ligne de ses déplacements
turtle.begin_fill()          # va remplir l'intérieur de ce qui est tracé entre
                             # maintenant et le turtle.end_fill() ultérieur
turtle.forward(300)           # la tortue avance de 300 (à droite)
turtle.right(144)             # la tortue effectue une rotation de 144° à droite
turtle.forward(300)           # avance dans la nouvelle direction
turtle.right(144)             # rotation
turtle.forward(300)           # avance
turtle.right(144)             # ...
turtle.forward(300)           #
turtle.right(144)             #
turtle.forward(300)           #
turtle.right(144)             #
turtle.end_fill()             # remplit ce qui a été tracé entre le begin_fill
                             # et cette instruction
turtle.hideturtle()          # cache la tortue
turtle.done()                # laisse l'utilisateur fermer la fenêtre

```

2.6.3 Premier pavé pour le projet Vasarely

PREMIER PAVÉ POUR LE PROJET VASARELY

Lancez-vous avec `turtle` : faites fonctionner votre imagination pour dessiner des figures. Vous pouvez aussi dessiner des polygones réguliers à 3, 5, 7... côtés.

Si vous avez besoin de vous rafraîchir la mémoire sur certaines notions de vos cours de mathématiques, une rapide recherche sur le Web vous permettra de (re)découvrir que pour dessiner un polygone à n côtés (par exemple $n = 5$), l'angle de la rotation à réaliser entre deux côtés est de $360^\circ/n$. Si n vaut 5 cela donne donc $360^\circ/5 = 72^\circ$.

Si l'on veut dessiner des étoiles à n branches et en supposant n impair (pour n pair, c'est un peu plus délicat) au moins égal à 5, l'angle intérieur sera de $(n-1)*180^\circ/n$ (par exemple 144° pour n valant 5 comme donné dans mon code précédent).

Un exemple de pavage hexagonal, comme nous le ferons dans le projet Vasarely, est la composition de trois losanges de couleur différentes comme ici en noir, bleu et rouge. Visionnez la vidéo pour comprendre la séquence réalisée par le code.

Note : Voir la vidéo de la section 2.6.3 : dessin d'un pavé hexagonal avec `turtle`

Pourquoi ne pas essayer : dessinez avec `turtle` un tel pavage en partant du point (0, 0) qui est la position initiale de la tortue, avec des lignes de longueur 100 par exemple.

- D'abord en utilisant en particulier les instructions `turtle.forward()`, `turtle.left()` et `turtle.right()` (Un petit conseil : utilisez des angles de 60° ou de 120° pour faire tourner votre tortue ; voir la figure « Géométrie d'un hexagone »)
- Ensuite, encore mieux, utilisez l'instruction `turtle.goto()` pour vos déplacements en calculant les abscisses et ordonnées des points extrémités où doit se déplacer la tortue ; voir les deux figures « Géométrie d'un hexagone » et « Rappel sur les sinus et cosinus ». Une des figures vous rappelle comment vous pouvez calculer ces coordonnées avec les fonctions sinus et cosinus (`math.sin` et `math.cos` après avoir importé le module `math`) : dans un espace à 2 dimensions, pour un cercle de centre (0,0) et de rayon 1, les coordonnées du point sur le cercle ayant un angle α avec l'axe des x (le point en

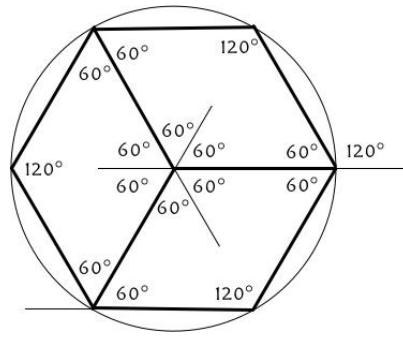


Fig. 2.1 – Géométrie d'un hexagone

rouge) est donné par $(\cos(\alpha), \sin(\alpha))$. Dans la figure « Rappel sur les sinus et cosinus », l'angle α vaut 60 degrés soit $\pi/3$ radians.

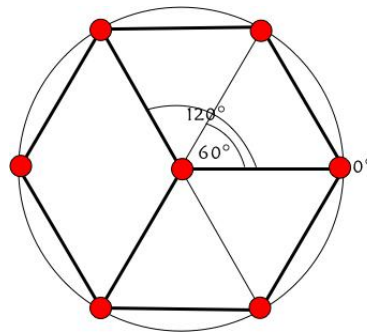


Fig. 2.2 – Géométrie d'un hexagone

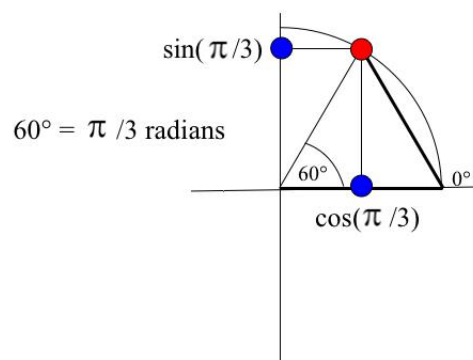


Fig. 2.3 – Rappel sur les sinus et cosinus

Avant de réaliser le pavage hexagonal nous vous demandons de réaliser l'exercice 2.4 du module 2 d'UpyLaB donné ci-après. La réussite de cet exercice vous permettra de réaliser plus facilement le pavage demandé ci-dessus.

Comme l'exercice contient un bagage mathématique en trigonométrie qui n'est pas l'objet de ce cours, nous proposons à ceux qui n'ont pas ou plus la connaissance des notions de sinus et cosinus d'avoir accès à une solution en bas de l'exercice UpyLaB 2.4. Notons que de ce fait, l'exercice n'est pas noté.

2.6.4 Exercice UpyLaB 2.4 - Non noté - Parcours Bleu et Rouge

Écrire un programme qui lit une longueur `long` de type float strictement positive, et qui affiche les valeurs `x` `y` des coordonnées (`x`, `y`) des sommets de l'hexagone de centre (0,0) et de rayon `long`

Chaque couple de coordonnées sera affiché sur une ligne différente, en commençant par le point à 0° , puis par le point à 60° , puis 120° ... jusqu'au 6ème point.

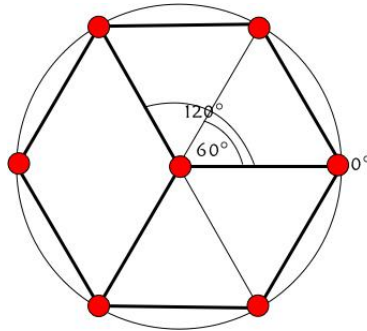


Fig. 2.4 – Géométrie d'un hexagone

Exemple 1

Avec la donnée lue suivante :

```
100.0
```

le résultat à imprimer vaudra (approximativement) :

```
100.0 0.0
50.0000000000000014 86.60254037844386
-49.999999999999998 86.60254037844388
-100.0 1.2246467991473532e-14
-50.000000000000004 -86.60254037844383
50.0000000000000014 -86.60254037844386
```

Exemple 2

Avec la donnée lue suivante :

```
5.5
```

le résultat à imprimer vaudra (approximativement) :

```
5.5 0.0
2.7500000000000004 4.763139720814412
-2.7499999999999987 4.763139720814413
-5.5 6.735557395310443e-16
-2.75000000000000027 -4.7631397208144115
2.7500000000000004 -4.763139720814412
```

Consignes

- Pour les calculs utilisez la valeur `pi` et les fonctions `sin` et `cos` du module `math`, comme indiqué sur la figure « Coordonnées d'un sommet de l'hexagone ».

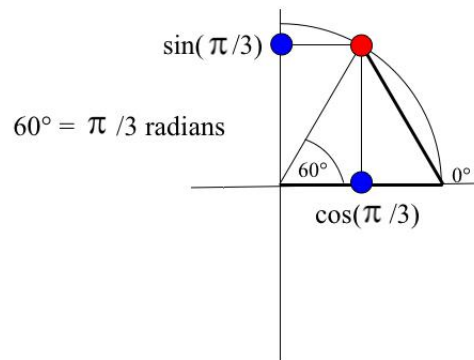


Fig. 2.5 – Coordonnées d'un sommet de l'hexagone

- Notez qu'il n'est pas demandé de tester si la valeur lue est bien strictement positive ; nous vous assurons que ce sera le cas pour les valeurs passées au programme dans les tests d'UpyLaB.
- Il ne vous est **pas** demandé de tracer l'hexagone, mais uniquement d'afficher des valeurs. Notez qu'UpyLaB ne supporte pas le module `turtle` et donc ne l'importez pas dans votre code pour cet exercice !
- Vous constaterez que l'utilisation des fonctions `sin()` et `cos()` et les erreurs d'arrondis et de troncature durant les calculs induiront que les résultats peuvent être un peu différents du résultat théorique ; Pour l'exemple 1 plus haut, les résultats plus précis devraient être :

```
100.0    0.0
50.0     86.6025403784438
-50      86.6025403784438
-100.0   0.0
-50.0    -86.6025403784438
50.0     -86.6025403784438
```

où 86.6025403784438 est une valeur approchée de la valeur de $100 \cdot \frac{\sqrt{3}}{2}$.

UpyLaB laisse une certaine latitude (ici de l'ordre de $1.0e-5$).

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

HAIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test un nombre différent en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes.
- Veillez à ce que votre affichage corresponde bien à ce qui est attendu ; `(1.0, 0.0)` n'est pas la même chose que `1.0 0.0`. Utilisez deux variables distinctes pour chacune des coordonnées et imprimez-les grâce à l'instruction `print(x, y)`.
- Si rien ne marche : consultez la [FAQ sur UpyLaB 2.4](#). Vous y trouverez en particulier quelques indices sur les calculs à effectuer.

Et si vraiment, vous ne trouvez pas

Vous n'arrivez pas à réaliser l'exercice. Voici une solution pour le résoudre :

```
from math import pi, sin, cos

long = float(input())
print(long * cos(0), long * sin(0))
print(long * cos(pi / 3), long * sin(pi / 3))
print(long * cos(pi * 2 / 3), long * sin(pi * 2 / 3))
print(-long, 0.0)
print(long * cos(4 / 3 * pi), long * sin(4 / 3 * pi))
print(long * cos(5 / 3 * pi), long * sin(5 / 3 * pi))
```

2.7 Pour terminer ce module

2.7.1 Stockage des valeurs et caractères d'échappement

PETITES REMARQUES SUR LES TROIS TYPES DÉJÀ VUS

Dans ce cours, nous ne voulons pas vous donner d'informations inutiles pour votre apprentissage. Par exemple, il est à ce stade inutile de trop détailler comment l'interpréteur Python réalise son travail. Il est pourtant important d'avoir certains éléments qui vont vous permettre de mieux comprendre le pourquoi des choses, et ainsi de savoir à quoi il faut faire attention dans vos codes.

Ainsi, pourquoi Python distingue-t-il les valeurs entières et fractionnaires ?

La réponse est dans le stockage en mémoire de l'ordinateur.

Entiers (int)

Tant que l'interpréteur Python a de la place mémoire disponible, il stocke des entiers aussi grands que demandé.

Si l'on demande à l'interpréteur Python dans PyCharm

```
>>> 3**100
>>> 3**1000
>>> 3**10000
```

les calculs peuvent prendre du temps mais vous pouvez vérifier que l'interpréteur donne le résultat.

Par contre, les valeurs entières ne stockent bien évidemment pas les parties fractionnaires ; ainsi

```
>>> 1 // 3
```

ou même

```
>>> 1 // 3 * 3
```

donnent la valeur 0.

Fractionnaires (float)

Pour les valeurs `float`, l'interpréteur fait des calculs et encode les valeurs dans un mot mémoire généralement de 64 bits, avec la meilleure précision qu'il peut et en retenant également le signe et une partie exposant pour pouvoir représenter des petits ou des grands nombres (grosso modo jusqu'environ $10e-300$ pour le plus petit nombre strictement positif et $10e300$ pour le plus grand).

Pour comprendre comment cela se passe, un parallèle peut être fait avec la représentation décimale des nombres fractionnaires :

si on doit représenter $1/3$ en nombre décimal avec un nombre fixé, par exemple 5, de chiffres après le point décimal, cela donne 0.33333 et on perd de la précision.

La représentation de nombres float en Python est similaire sauf que tout est représenté en binaire, c'est-à-dire avec comme seuls chiffres des 0 et des 1.

Nous vous invitons à tester cela vous même avec PyCharm en tapant par exemple :


```
>>> 1.0e308
>>> 1.0e309
>>> 1.0e-323
>>> 1.0e-324
>>> 1.00000000000000000001
```

pour voir ce que l'interpréteur donne comme valeurs.

En général, on perd donc de la précision dans les calculs et parfois les calculs ne peuvent être simplement réalisés avec les valeurs manipulées en tant que valeurs de type float.

Par exemple, à la place de 1.0e309, l'interpréteur affiche la valeur `inf` : il s'agit de l'infini qui marque que la valeur est trop grande pour être manipulée avec des float. Si vous désirez manipuler de telles valeurs dans vos codes Python, il faudra trouver des types plus étendus pour le faire. C'est possible mais sort du cadre de ce cours.

Pour ceux qui veulent faire des calculs scientifiques précis et désirent une explication plus complète ainsi que des alternatives pour faire mieux, allez voir sur le site python.org, dans la documentation Python 3 et cherchez "Floating Point Arithmetic : Issues and Limitations".

Chaînes de caractères avec caractères d'échappement

Il se peut que nous voulions imprimer du texte avec des tabulations, passages à la ligne ou autres caractères spéciaux comme des quotes simple ou doubles à l'intérieur du texte.

Pour cela, on peut utiliser le caractère d'échappement (appelé "escape" en anglais).

Par exemple

```
print('\n\nBonjour\n\n')
```

demande d'imprimer le texte qui contient au début 2 passages à la ligne suivi du texte Bonjour et terminé par à nouveau 2 passages à la ligne (soit 11 caractères au total qui seront imprimés sur 4 lignes).

Le tableau suivant donne quelques caractères spéciaux dénotés grâce au caractère d'échappement. Notez que de ce fait, pour spécifier qu'un caractère antislash fait partie du texte, il faut mettre deux antislashes : la première exprimant que ce qui suit est le caractère antislash lui-même.

```
print('\\')
```

imprime juste un seul caractère antislash.

Notez que si le caractère qui suit l'antislash dans un texte n'est pas reconnu comme caractère spécial, l'antislash est interprétée littéralement par l'interpréteur python.

Ainsi `print('\B')` imprime bien les deux caractères `\B`.

<code>\'</code>	La simple quote (')
<code>\"</code>	La double quote (")
<code>\n</code>	Le passage à la ligne ASCII
<code>\t</code>	La tabulation horizontale
<code>\\</code>	L'antislash (backslash \)

N'hésitez pas à tester ces caractères par vous-même pour être sûr que vous les maîtrisez.

Note : Chaîne de caractères multiligne

Nous avons vu qu'au début de nos scripts, il était recommandé de mettre un commentaire multiligne donnant des informations sur ce que fait ce script, son auteur, la date de création et éventuellement de dernière modification.

Un commentaire multiligne est en fait une chaîne de caractères multiligne qui peut être manipulée comme n'importe quelle autre chaîne de caractères. Sa particularité est qu'il peut s'étaler sur plusieurs lignes ; chaque passage à la ligne correspondant au caractère dénoté `\n`.

Par exemple, le code

```
mon_texte = '\n\nBonjour\n\n'
```

peut être écrit :

```
mon_texte = """
Bonjour
"""
```

Dans ce cas, faites attention à l'indentation puisque ici, tout ce qui est entre les trois doubles quotes fait partie du texte.

2.7.2 Les opérateurs d'assignation et de mise à jour

COMPLÉMENT SUR L'ASSIGNATION

Pour être précis l'interpréteur Python manipule des objets contenant les valeurs et des variables qui sont des noms pour les objets. Notons que l'instruction d'assignation multiple

```
x = y = 3
```

crée un objet de type entier contenant la valeur 3 ; `x` et `y` sont deux noms donnés à cet objet.

Si juste après, l'interpréteur exécute

```
x = 4
```

après cette instruction, la variable `x` aura changé de valeur et vaudra 4 (`x` désigne la valeur 4), tandis que `y` reste inchangée et a toujours la valeur 3.

Les opérateurs de mise à jour

Python permet d'écrire de façon courte des opérations qui consistent à prendre la valeur d'une variable et de lui appliquer une opération avec une valeur comme deuxième opérande.

L'exemple le plus courant est l'incrémentation :

supposons que `x` soit égal à 5 (par exemple après l'assignation `x = 5`)

```
x += 1
```

incrémente `x` de 1, lui donnant la valeur 6.

Attention

```
x = 5
x += 5 * 7
```

Le code ci-dessus assigne 5 à la variable `x`, ensuite évalue `5 * 7` avant d'incrémenter `x` du résultat. C'est donc l'équivalent de `x = x + (5 * 7)`.

Ce principe peut être appliqué avec tous les opérateurs arithmétiques (`+=`, `-=`, `*=`, `/=`, `//=`, `**=`, `%=`).

2.7.3 On passe à la pratique autonome !

MISE EN PRATIQUE AVEC UPYLAB

Comme déjà annoncé, on ne peut devenir un bon codeur sans la mise en pratique répétée des concepts vus.

C'est le rôle d'UpyLaB de vous obliger à réaliser cette pratique.

Nous vous demandons de réaliser l'ensemble des exercices du module 2 qu'il vous reste à faire et qui sont donnés dans les pages suivantes de cette section.

En cas de difficultés, nous vous rappelons ces quelques conseils :

1. Relire l'énoncé et les consignes associés à l'exercice ainsi que les éventuels conseils.
2. Lire la FAQ générale à propos des exercices UpyLaB.
3. Lire, si elle existe, la FAQ spécifique à l'exercice réalisé.
4. Si rien de tout cela ne fonctionne, poser votre question sur le Forum dédié aux exercices, en étant le plus précis possible, *sans mettre votre code complet ni substantiel* mais en mettant des copies d'écran des résultats fournis par UpyLaB.

2.7.4 Exercice UpyLaB 2.5 - Parcours Vert, Bleu et Rouge

Énoncé

Le but de cet exercice est de vous familiariser avec la syntaxe Python pour écrire des expressions arithmétiques simples et avec l'instruction `print` qui affiche (on dit aussi imprime) des valeurs à l'écran.

Écrire un programme qui lit deux valeurs entières x et y strictement positives suivies de deux valeurs réelles (float) z et t , et qui affiche les valeurs des expressions suivantes, chacune sur une nouvelle ligne :

- $x - y$
- $x + z$
- $z + t$
- $x.z$ (produit de x et de z)
- $\frac{x}{2}$
- $\frac{x}{y+1}$
- $\frac{(x+y).z}{4.x}$
- $x^{-\frac{1}{2}}$ (x exposant $-\frac{1}{2}$)

Exemple

Avec les données lues suivantes :

```
2
1
3.0
3.5
```

le résultat à imprimer vaudra (approximativement pour la dernière valeur) :

```
1
5.0
6.5
6.0
1.0
1.0
1.125
0.7071067811865476
```

Consignes

- Il n'est pas demandé de tester si les valeurs de x et de y sont bien strictement positives.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.5*.

2.7.5 Exercice UpyLaB 2.6 - Parcours Vert, Bleu et Rouge

Énoncé

Le but de cet exercice est de vous familiariser avec l'impression de certains caractères particuliers comme les simples et double quotes (apostrophes), l'anti-slash » (appelée également la barre oblique inversée `\`), ...

Écrire un programme affichant les quatre lignes suivantes :

```
Hello World
Aujourd'hui
C'est "Dommage !"
Hum \o/
```

Consignes

Veillez à ce que votre programme n'affiche rien de plus que ce qui est attendu : pas d'espace en fin de ligne, pas de ligne vide supplémentaire, ..., et faites attention aux détails : respectez la ponctuation (les simples et doubles quotes sont droites), le nombre d'espaces entre les mots...

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Si votre code n'est pas accepté et qu'il semble pourtant afficher ce qui est attendu, c'est qu'il peut y avoir des espaces superflues en fin de ligne. Attention en particulier à la fonction `print`, qui ajoute par défaut une espace entre ses arguments s'il y en a plusieurs. Une solution consiste à modifier l'argument nommé `sep` de cette fonction, en lui attribuant par exemple la valeur chaîne vide (`sep = ""`), ou encore à utiliser plusieurs appels à cette fonction.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.6*.

2.7.6 Exercice UpyLaB 2.7 - Parcours Bleu et Rouge

Écrire un programme qui imprime la valeur du volume d'une sphère de rayon r , float lu en entrée.

On rappelle que le volume d'une sphère de rayon r est donné par la formule : $\frac{4}{3}\pi r^3$

Exemple 1

Avec la donnée lue suivante :

```
1.0
```

le résultat à imprimer vaudra (approximativement) :

```
4.1887902047863905
```

Exemple 2

Avec la donnée lue suivante :

```
0.5
```

le résultat à imprimer vaudra (approximativement) :

```
0.5235987755982988
```

Consignes

- Il n'est pas demandé de tester si la valeur lue en entrée est bien positive ou nulle.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Pour rappel, en Python, l'opérateur exposant est `**`. Ainsi, `2 ** 3` vaut 8 soit `2 * 2 * 2`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.7*.

2.8 Quiz de fin et bilan du module

2.8.1 Quiz de fin de module

MINI QUIZ

Note : Voir le quiz de la section 2.8.1

BILAN EN BREF

Note : Voir la vidéo de la section 2.8.2 : Bilan du module 2

BILAN DU MODULE

Nous voici à la fin de ce module. Nous avons déjà vu beaucoup de concepts de base Python 3 dont voici les principaux :

- Nous avons vu que l’interpréteur Python 3 peut être utilisé en mode interactif ou en mode script en particulier dans l’environnement PyCharm.
- Nous avons vu qu’un code Python peut utiliser des valeurs et expressions entières (int), fractionnaires (float) ou chaînes de caractères (str pour string), la fonction prédéfinie `type(x)` donnant le type de `x`.
- Votre code peut être déclaré incorrect parce qu’il contient des erreurs syntaxiques.
- Nous avons également vu que Python peut se créer des variables ayant des valeurs grâce à l’instruction d’*assignation* (appelée également *affectation*).
- Les instructions `input` et `print` permettent au script Python de “recevoir des données” de l’utilisateur et d’ “imprimer des résultats”.
- Pour avoir un code plus lisible pour le programmeur, des commentaires simples et multilignes peuvent être écrits dans le code.
- Python offre des fonctions prédéfinies (`type()`, `int()`, `float()`, `dir()` et `help()` et `max()`, `round()`, `divmod()`...), ainsi que des modules comme `math` et `turtle`, utilisables via le verbe `import`.
- Nous avons vu que les diagrammes d’état expliquent de façon graphique comment fonctionne un code Python et que l’outil Python Tutor (accessible via la page web pythontutor.com) permet de visualiser ces diagrammes d’état en exécutant pas à pas les petits scripts Python fournis.
- Nous avons également illustré tous ces concepts avec de nombreux exemples et la réalisation d’exercices supervisés ou réalisés de façon autonome avec UpyLaB.

Les instructions : tous vos désirs sont des ordres

3.1 Presque toutes les instructions au menu

3.1.1 Présentation du menu de ce module

LES INSTRUCTIONS PYTHON

La vidéo qui suit présente le menu du présent module : l'étude des instructions de contrôle de flux Python. Elles sont appelées instructions de contrôle de flux parce qu'elles permettent de rompre la simple séquence d'exécution comme ce que nous avons vu jusqu'à présent. Nous verrons que ces instructions sont essentielles pour rédiger du code Python.

MENU DE CE MODULE

Note : Voir la vidéo de la section 3.1.1 : Menu du module 3

3.2 L'instruction conditionnelle if : fais pas « si », fais pas ça !

3.2.1 L'instruction if

INTRODUCTION À L'INSTRUCTION CONDITIONNELLE IF

La présente section vous explique tout ce qui vous sera nécessaire sur l'instruction conditionnelle `if`.

Cette vidéo introduit l'instruction `if` qui permet d'exécuter certaines instructions uniquement si une condition est vérifiée.

Comme le traitement va dépendre d'une condition, nous parlerons de l'instruction conditionnelle `if`.

Nous partons d'un exemple simple avec seulement le mot-clé `if` pour expliquer le fonctionnement de cette instruction et petit à petit, nous ajoutons des éléments, en particulier qui utilisent les mots-clés `elif` et `else`, pour montrer différentes possibilités.

PRÉSENTATION DE L'INSTRUCTION IF

Note : Voir la vidéo de la section 3.2.1 : L'instruction if

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction conditionnelle `if` et ses formats possibles : sans parties `else` et `elif`, avec une partie `else` et avec une partie `elif`.

Scripts et code sur la console réalisés dans la vidéo

```
maximum = 0
releve = int(input())
if releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
print("Maximum retenu :", maximum)
```

```
>>> 3 < 5
True
>>> 5 < 3
False
>>> type(False)
<class 'bool'>
>>> type(True)
<class 'bool'>
>>> type(3<5)
<class 'bool'>
>>> x = int(input())
>? 5
>>> if x < 0:
...     print("ok")
...
>>>
```

```
maximum = 10
releve = int(input())
if releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
else:
    print("Pas de nouveau record")
print("Maximum retenu :", maximum)
```

```
maximum = 10
releve = int(input())
if releve == 0:
    print("Pas de pluie aujourd'hui")
elif releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
else:
    print("Pas de nouveau record")
print("Maximum retenu :", maximum)
```


La vidéo en bref

L'instruction conditionnelle `if` est présentée grâce à des scripts. Sa forme la plus simple est :

```
if condition:
    instructions
```

où la condition peut par exemple être une comparaison entre deux valeurs en utilisant un opérateur relationnel tel que :

- *est strictement inférieur* `<` ou
- *est strictement supérieur* `>`.

Cette comparaison donne une valeur de type *booléenne* vraie (`True`) ou fausse (`False`).

L'indentation permet d'identifier les instructions dans le `if`.

Un `else` peut être ajouté à la fin de l'instruction `if` et une ou plusieurs parties `elif` peuvent être également ajoutées comme montré dans la vidéo.

L'instruction `if` avec parties `elif` et `else` a la forme :

```
if condition:
    instructions
elif condition:
    instructions
elif condition:
    instructions
...
else:
    instructions
```

Dans tout les cas, au maximum une seule « branche » est exécutée, et elle correspond au premier test évalué à vrai soit au niveau du `if` soit au niveau d'un `elif` et sinon, au niveau du `else` s'il existe. Ensuite l'interpréteur Python passe à l'instruction suivante.

Si tous les tests du `if` et `elif` sont évalués à faux et qu'il n'y a pas de partie `else`, aucune « branche » du `if` ne sera exécutée.

3.2.2 Les opérateurs relationnels

LES OPÉRATEURS RELATIONNELS

Nous avons vu qu'un élément clé de l'instruction `if` est la condition associée, dont la valeur, vraie (`True`) ou fausse (`False`), est souvent le résultat de comparaisons entre valeurs. Donnons ici tous les opérateurs de comparaison, appelés opérateurs relationnels Python.

Supposons que les variables `a` et `b` aient chacun une certaine valeur (par exemple `a = 3` et `b = 5`). Les opérateurs de comparaisons, appelés dans le jargon informatique opérateurs relationnels, possibles sont :

<code>a < b</code>	la valeur de <code>a</code> est-elle strictement inférieure à la valeur de <code>b</code>
<code>a > b</code>	la valeur de <code>a</code> est-elle strictement supérieure à la valeur de <code>b</code>
<code>a <= b</code>	la valeur de <code>a</code> est-elle inférieure ou égale à la valeur de <code>b</code>
<code>a >= b</code>	la valeur de <code>a</code> est-elle supérieure ou égale à la valeur de <code>b</code>
<code>a == b</code>	la valeur de <code>a</code> est-elle égale à la valeur de <code>b</code>
<code>a != b</code>	la valeur de <code>a</code> est-elle différente de la valeur de <code>b</code>

Attention : à ne pas confondre ou utiliser l'assignation `=` à la place de l'opérateur relationnel d'égalité `==`.

3.2.3 Exemples de code avec des if

ANNÉE BISSEXTILE

Continuons ici d'illustrer l'utilisation du `if`, sur un exemple concret : un code qui reçoit une année et affiche si celle-ci est ou non bissextile (c'est-à-dire qui comprend 366 jours y compris un 29 février).

Version simple sans elif

Dans une première approximation, nous considérons que nous avons une année bissextile tout les quatre ans et que l'année bissextile est celle dont la valeur est divisible par 4 (par exemple 2020 est bissextile).

Un nombre x est divisible par 4 si le reste de la division entière de x par 4 vaut zéro. En Python cela se teste aisément avec `x % 4 == 0`.

Le premier code complet est donc :

```
annee = int(input("Donnez l'année à tester :"))
if annee % 4 == 0:
    print("l'année", annee, "est bissextile")
else:
    print("l'année", annee, "n'est pas bissextile")
```

Exemple d'instruction if avec elif et else

En consultant l'entrée [année bissextile sur wikipédia](#), nous voyons que la définition est plus précise.

Depuis l'ajustement du calendrier grégorien, l'année sera bissextile :

- si l'année est divisible par 4 et non divisible par 100, ou
- si l'année est divisible par 400.

Sinon, l'année n'est pas bissextile.

Observons sur l'animation Python Tutor ci-dessous que le script donne bien le bon résultat avec par exemple 2020 et 2000 qui sont bissextiles et 2100 et 2019 qui ne le sont pas (à chaque test n'oubliez pas de recharger la page sinon Python Tutor utilise la même valeur d'input). Avec l'animation, observons également le fonctionnement de l'instruction `if`, en particulier quelles sont les instructions exécutées.

Note : Animation voir cours section 3.2.3

```
annee = int(input())
if annee % 400 == 0:
    print('bissextile')
elif annee % 100 == 0:
    print('non bissextile')
elif annee % 4 == 0:
    print('bissextile')
else:
    print('non bissextile')
```

CALCUL DES RACINES D'UNE ÉQUATION DU SECOND DEGRÉ

Donnons un autre exemple plus mathématique cette fois : analysons comment écrire un code qui calcule la ou les éventuelles racines d'une équation du second degré.

Note : Ceux qui ne désirent pas faire des mathématiques maintenant peuvent simplement ignorer cet exemple qui ne donne pas de nouvelles notions Python.

Expliquons ou rappelons le problème et comment le résoudre.

Rappel : ayant l'équation du second degré

$$ax^2 + bx + c = 0$$

où a , b et c sont des valeurs réelles, pour savoir si cette équation a deux, une ou aucune racines réelles, il faut calculer le *discriminant* $delta$ qui vaut :

$$delta = b^2 - 4ac$$

1. Si la valeur de $delta$ est strictement positive, l'équation a deux racines réelles :

$$\begin{aligned} \text{--- } x_1 &= \frac{-b - \sqrt{delta}}{2a} \\ \text{--- } x_2 &= \frac{-b + \sqrt{delta}}{2a} \end{aligned}$$

2. Si la valeur de $delta$ est égale à 0, l'équation a une racine :

$$\text{--- } x = \frac{-b}{2a}$$

3. Si la valeur de $delta$ est strictement négative, l'équation n'a pas de racine réelle.

Le script animable suivant propose une solution au problème. Notez l'indentation des instructions qui correspondent respectivement aux parties `if`, `elif` et `else`, ainsi que l'utilisation de l'opérateur exposant un demi (par exemple : `delta ** 0.5`) pour calculer la racine carrée d'une valeur.

Animation Python Tutor de l'exemple

Testons-le avec une animation Python Tutor par exemple avec comme valeurs pour a , b , c :

- 1, -3, 2, ce qui correspond à l'équation $(x - 1).(x - 2) = 0$
- 1, -2, 1, ce qui correspond à l'équation $(x - 1)^2 = 0$
- 1, 1, 1, ce qui correspond à l'équation $x^2 + x + 1 = 0$, où le delta est négatif (-3)

Important : rechargez la page entre chaque test pour réinitialiser l'exécution.

Note : Voir la seconde animation en section 3.2.3

Code de l'animation

```
a= float(input('a : '))
b= float(input('b : '))
c= float(input('c : '))

delta = b**2 - 4*a*c

if delta < 0:
    print(" pas de racines réelles")
elif delta == 0:
    print("une racine : ")
    print("x = ", -b/(2*a))
else:
    racine = delta**0.5
    print("deux racines : ")
    print("x1 = ", (-b + racine)/(2*a))
    print("x2 = ", (-b - racine)/(2*a))
```

3.2.4 Opérateurs logiques

OPÉRATEURS LOGIQUES AND, OR ET NOT

Continuons à compléter les possibilités d'une instruction `if`. Plus précisément regardons les opérateurs qui peuvent être présents dans une condition associée à un `if` ou un `elif`.

La condition d'un `if` est une expression booléenne qui peut utiliser la valeur `True` ou `False`, ou une comparaison de valeurs avec des opérateurs relationnels, ou également les opérateurs logiques `and`, `or` et `not`.

Regardons sur quelques exemples comment fonctionnent les opérateurs logiques, appelés aussi opérateurs booléens, en supposant que la variable `x` contient une valeur entière.

- `0 <= x and x < 10` : teste si la valeur de `x` est dans l'intervalle `[0 .. 10[` c'est-à-dire la valeur de `x` est supérieure ou égale à 0 et aussi strictement inférieure à 10.
- `flag = x < 0 or 10 <= x` : assigne à la variable `flag` une valeur logique vraie si la valeur de `x` est soit strictement inférieure à 0 soit supérieure ou égale à 10, et fausse sinon.
- `not (x < 0)` : est équivalent à `x >= 0`.

Notez que :

```
if x % 2 == 0 or x % 3 == 0:
    print('x pair ou multiple de 3')
```

pour `x` valant 6, les deux tests sont vrais et donc le résultat du `or` aussi :

on parle du « ou inclusif » voulant dire que si une des deux ou si les deux parties sont vraies, le résultat est également vrai. Pour comprendre plus en détail comment l'interpréteur évalue les expressions booléennes utilisant des opérateurs logiques `and`, `or` ou `not`, il faut connaître la table de vérité de ces opérateurs.

Table de vérité du `not`, `and` et `or`

La table de vérité des opérateurs `and` (qui correspond au « et » en français) `or` (« ou » en français) et `not` (« non » en français) définit la sémantique de chacun des trois opérateurs logiques.

La table doit être lue comme suit : supposons que l'on ait une expression contenant des sous-expressions dont la valeur est logique (Vraie ou Fausse). Ici, les sous-expressions sont dénotées par `a` et `b`.

En fonction des différentes valeurs possibles de `a` et de `b`, Fausse (`False`) ou Vraie (`True`), la table donne la valeur de chacune des expressions `not a`, `a and b`, `a or b`.

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Associativité et priorité des opérateurs relationnels et logiques

Complétons le tableau donnant l'associativité et la priorité des opérateurs en ajoutant les opérateurs relationnels et logiques, depuis le plus prioritaire vers le moins prioritaire :

(expression)	
<code>**</code>	associatif à droite
<code>*</code> <code>/</code> <code>//</code> <code>%</code>	associatifs à gauche
<code>+</code> <code>-</code>	associatifs à gauche
<code><</code> <code><=</code> <code>></code> <code>>=</code> <code>!=</code> <code>==</code>	associatifs à gauche
<code>not x</code>	
<code>and</code>	associatif à gauche
<code>or</code>	associatif à gauche

où les opérateurs sur une même ligne ont le même niveau de priorité.

Lois de De Morgan

Les lois de De Morgan sont fréquemment utilisées pour simplifier les tests ; ayant des expressions logiques `a` et `b`

- `not (a or b)` est équivalente à `(not a) and (not b)` ;
- `not (a and b)` est équivalente à `(not a) or (not b)`.

Ainsi :

```
not(0 <= x and x < 10)
```

est équivalent à

```
(not(0 <= x)) or (not(x < 10))
```

qui par ailleurs vaut, en prenant les opérateurs duaux de `<=` et de `<` (sachant que `>` est l'opérateur dual de `<=` et vice versa) :

```
0 > x or x >= 10
```

L'EXEMPLE DE L'ANNÉE BISSEXTILE REVISITÉ

Grâce à l'utilisation des opérateurs logiques, nous pouvons maintenant produire un code plus simple du script qui affiche si une année lue en input est bissextile.

En effet, comme il n'y a que deux cas possibles (soit l'année est bissextile soit elle ne l'est pas) il est logique d'avoir une simple instruction `if` qui teste par exemple tous les cas où l'année est bissextile suivie d'un seul `else` dans le cas contraire.

Voici un exemple de code pour cela :

```
annee = int(input("Donnez l'année à tester :"))
if (annee % 4 == 0 and annee % 100 != 0) or annee % 400 == 0:
    print('bissextile')
else:
    print('non bissextile')
```

Notez que les parenthèses ont été mises pour clarifier la lecture du code, mais vu que le `and` est plus prioritaire que le `or`, elles ne sont pas nécessaires.

3.2.5 Syntaxe et sémantique de l'instruction if

SYNTAXE GÉNÉRALE DU IF

Maintenant que nous avons introduit progressivement à partir d'exemples comment fonctionne l'instruction `if`, découvrons sa forme générale. Effectivement, avant de pouvoir utiliser l'instruction `if` totalement à bon escient, il faut connaître ses syntaxe et sémantique ainsi qu'un certain nombre de règles que nous vous livrons dans cette activité.

La syntaxe générale de l'instruction `if`, telle que vous la trouverez dans les manuels de référence Python, est donnée par :

```
"if" expression ":" suite
("elif" expression ":" suite)*
["else" ":" suite]
```

où :

- `"if"`, `"elif"`, `"else"` sont des mots-clés; les doubles apostrophes ne doivent pas être écrites,
- `expression` est une expression booléenne,
- le caractère `" : "` sans les doubles apostrophes est mis après `expression` à la ligne du `if` ou des `elif` ou après le mot-clé `else`,
- `suite` est un bloc d'instructions indentées, c'est-à-dire décalées par exemple de 4 caractères vers la droite mises sur la ou les lignes suivantes,
- `(...) *` est une méta-notation signifiant que la ligne peut ne pas être mise ou être mise une ou plusieurs fois,
- `[...]` est une méta-notation signifiant que la ligne est optionnelle, c'est-à-dire peut ou non être mise.

MOTS-CLÉS ET IDENTIFICATEURS

Dans les présentations précédentes de ce module, nous avons parlé de mots-clés. Avant de continuer, il est important d'expliquer un peu plus le but de ces mots-clés et ce à quoi il faut faire attention.

Mots-clés

Les mots-clés (keywords en anglais) Python caractérisent certains éléments du langage. Par exemple, le mot-clé `if` exprime que ce qui suit est une instruction conditionnelle. Ils doivent être écrits tel que spécifié; par exemple pour le mot-clé `if`, les caractères doivent être des minuscules.

Python possède une trentaine de mots-clés dont la liste (pour Python 3.7) est la suivante :

<code>False</code>	<code>async</code>	<code>del</code>	<code>from</code>	<code>lambda</code>	<code>return</code>
<code>None</code>	<code>await</code>	<code>elif</code>	<code>global</code>	<code>nonlocal</code>	<code>try</code>
<code>True</code>	<code>break</code>	<code>else</code>	<code>if</code>	<code>not</code>	<code>while</code>
<code>and</code>	<code>class</code>	<code>except</code>	<code>import</code>	<code>or</code>	<code>with</code>
<code>as</code>	<code>continue</code>	<code>finally</code>	<code>in</code>	<code>pass</code>	<code>yield</code>
<code>assert</code>	<code>def</code>	<code>for</code>	<code>is</code>	<code>raise</code>	

Nous en avons déjà vu certains, comme `import`, `if`, `elif`, `else`, `and`, `or`,...

Nous en verrons d'autres dans ce cours, mais pas tous. Certains correspondent en effet à des concepts avancés.

Identificateurs

En dehors des mots-clés, un programme Python utilise entre autres des variables et des fonctions. Chaque variable et fonction porte généralement un nom. Par exemple dans les exemples précédents nous avons `x`, `a` et `delta` comme noms de variables ou `int`, `input`, `print` comme noms de fonctions prédéfinies ou de type.

Dans le jargon informatique, ces noms de variables, fonctions, types sont appelés identificateurs. Comme son nom l'indique, un identificateur est le nom qui identifie quelque chose. Un identificateur n'est pas un mot-clé.

Comme les mots-clés sont des mots réservés pour dénoter quelque chose de précis dans le langage, il ne peuvent évidemment pas être utilisés à d'autres fins. Ainsi, essayer d'appeler une variable avec le nom d'un mot-clé va générer une erreur lors de l'exécution. Par exemple si nous demandons à l'interpréteur d'exécuter

```
if = 5
SyntaxError: invalid syntax
```

il m'indique qu'il y a une erreur de syntaxe puisque l'interpréteur s'attend, après le mot-clé `if`, à avoir une expression booléenne et une suite bien précise et pas un symbole d'assignation comme ici.

Lors de la rédaction d'un code Python, nous devons donc bien faire attention à ne pas utiliser un mot-clé pour nommer par exemple une variable.

Notons que réutiliser, comme nom d'identificateur, le nom d'un identificateur prédéfini est possible même si c'est une très mauvaise idée. Ainsi on peut écrire :

```
print = 3
```

mais par la suite

```
print("Résultat =", res)
```

donnera une erreur puisque `print` ne correspond plus à la fonction prédéfinie, mais à une variable entière et donc lui accoler des parenthèses ne veut plus rien dire.

Ne faites donc pas cela !

N'oubliez pas l'indentation

Respectez parfaitement l'indentation des instructions Python, c'est-à-dire le fait d'écrire les lignes de code associées à un `if`, `elif` ou `else`, en les décalant de par exemple 4 caractères.

Faites-y toujours bien attention sachant que :

```
x = int(input('donnez une valeur entière : '))
if x > 0 :
    print('la valeur est strictement positive')
print('Au revoir')
```

où, quelle que soit la valeur lue, le message “Au revoir” est affiché à l'écran, est différent du code :

```
x = int(input('donnez une valeur entière : '))
if x > 0 :
    print('la valeur est strictement positive')
    print('Au revoir')
```

où le message “Au revoir” n'est affiché que si la valeur lue est strictement positive.

L'oubli ou une mauvaise indentation est une erreur très fréquente quand vous débutez en programmation Python. Cela peut générer des erreurs de syntaxe ou pire, des erreurs dans les instructions qu'il fallait exécuter pour que le code soit correct.

IF IMBRIQUÉS

Un code Python peut également avoir des `if` imbriqués par exemple pour distinguer des sous-cas. Il faut bien faire attention à l'indentation comme le montrent les deux bouts de code suivants :

Code 1

```
if a > 0 :
    if b > 0 :
        print("cas 1")
    else :
        print("cas 2")
```

où le `else` correspond au `if` imbriqué (celui le plus à l'intérieur) et

Code 2

```
if a > 0 :
    if b > 0 :
        print("cas 1")
else :
    print("cas 2")
```

où le `else` correspond au `if` global.

Pour comprendre finement ce que fait ce code avec différentes valeurs des variables `a` et `b`, n'hésitez pas à tester ces deux exemples dans Python Tutor.

Pour cela, vous pouvez exécuter étape par étape les deux codes des animations, générées par Python Tutor (comme habituellement avec Python Tutor, utilisez les boutons « back » et « forward » et communiquez les données dans la fenêtre de soumission puis cliquez sur « submit ») :

Animation Python Tutor du code 1

Note : Voir la première animation de la section 3.2.5 du cours en ligne

Animation Python Tutor du code 2

Note : Voir la seconde animation de la section 3.2.5 du cours en ligne

ÉVALUATION PARESSEUSE

Il nous reste à voir une dernière règle qui pourra s'avérer très utile sur l'évaluation des expressions booléennes. Python est paresseux quand il évalue certaines expressions.

En clair, en Python,

- si l'expression booléenne est un `and` et la valeur de gauche est évaluée à Faux, la partie à droite du `and` n'est jamais évaluée et la réponse au test complet est Faux ;
- de façon duale, si l'expression booléenne est un `or` et si la valeur de gauche est Vraie, la partie à droite du `or` n'est jamais évaluée et la réponse au test complet est Vrai.

L'évaluation paresseuse peut être utile si par exemple la première partie de la condition détermine si le second test peut être réalisé.

Par exemple : avec deux variables `a` et `b` contenant chacune une valeur entière positive, le code suivant est correct :

```
if a != 0 and b % a == 0:
    print(b, "est un multiple de", a)
```

Il teste d'abord la condition `a != 0` ; si ce test est faux, l'évaluation de la condition complète s'arrête et a la valeur `False`. Si par contre la condition `a != 0` a la valeur `True`, l'évaluation de la condition `b % a == 0` est effectuée par l'interpréteur.

Cette façon de faire permet donc de s'assurer que le calcul du modulo (`b % a`) ne va pas donner une erreur à l'exécution (division par zéro) et sinon de ne pas faire cette opération.

3.2.6 Quiz sur l'instruction if

MINI QUIZ SUR CE QUE L'ON A DÉJÀ VU

Note : Voir le quiz de la section 3.2.6

3.3 Pratique de l'instruction if

3.3.1 Mettons ensemble en pratique ce que nous venons de voir

UN PETIT JEU DE DEVINETTE

Maintenant que vous maîtrisez les subtilités de l'instruction `if`, passons à la pratique !

Écrivons d'abord un programme qui propose un petit jeu à l'utilisateur. Par la suite nous compléterons ce jeu et nous reviendrons à ce programme plus tard dans ce module afin d'illustrer d'autres instructions.

Commençons par une première version simplifiée dans laquelle :

Consignes : le programme :

- choisit aléatoirement un nombre entre 0 et 5 sans en afficher la valeur (et donc sans que l'utilisateur connaisse cette valeur) et le place dans la variable `secret` ;
- demande à l'utilisateur de deviner la valeur choisie ;
- affiche "gagné !" si l'utilisateur trouve la bonne réponse et
- affiche "perdu ! La valeur était " suivi de la valeur de `secret` dans le cas contraire.

Petite astuce pour vous permettre de vous lancer

En informatique, la génération de nombres aléatoires se fait généralement par une fonction qui fait des calculs en fonction de paramètres divers et produit un résultat dans l'intervalle demandé. On parle donc plutôt de génération de nombre pseudo aléatoire. En effet, si le nombre est vu par l'utilisateur comme étant aléatoire, il provient en réalité de calculs précis effectués par l'ordinateur.

En Python, le module `random` peut être utilisé à cette fin. En particulier `random` contient la fonction prédéfinie `randint(a, b)` où `a` et `b` sont des valeurs entières (par exemple 0 et 5). À chaque nouvel appel, `randint` génère un nombre pseudo aléatoire dans l'intervalle entre la valeur `a` et la valeur `b` toutes deux comprises (`[a, b]`).

Le code peut donc commencer par :

```
import random
secret = random.randint(0, 5)
```

À vous de jouer ! Écrivez un script dans PyCharm qui résout l'exercice proposé avec les consignes données ci-dessus.

3.3.2 Jeu de devinette : proposition de solution

PROPOSITION DE SOLUTION

Quand plusieurs personnes écrivent du code pour résoudre un problème, comme ici avec le petit jeu de devinette, les codes sont très certainement différents même s'ils peuvent tous parfaitement résoudre le problème. Malgré cela, il peut quand même être intéressant de comparer les solutions pour voir laquelle semble la plus claire ou la plus efficace. Ainsi, si vous voulez comparer votre solution avec un autre code qui résout le problème quand vous aurez terminé, ou si vous avez besoin d'un petit coup de pouce, la vidéo suivante propose une solution.

PROPOSITION DE SOLUTION

Note : Voir la vidéo de la section 3.3.2 : Proposition de solution

CONTENU DE LA VIDÉO

La vidéo précédente présente une solution du petit jeu de devinette. Le code d'une solution est donné plus bas.

CODE D'UNE SOLUTION

Ci-dessous la solution proposée dans la vidéo de la section 3.3.2.

```
""" Petit jeu de devinette (version 1)
Auteur: Thierry Massart
Date : 5 avril 2018
"""

import random
secret = random.randint(0,5)
choix_utilisateur = int(input("Donnez votre choix pour la valeur secrète : "))
if secret == choix_utilisateur:
    print("gagné !")
else:
    print("perdu ! La valeur était", secret)
print("Au revoir !")
```

3.3.3 Le if : mise en pratique autonome : exercice UpyLaB 3.1

MISE EN PRATIQUE AUTONOME

Nous voici arrivés à la troisième phase d'apprentissage en ce qui concerne l'instruction `if`. Dans l'activité qui va suivre, nous vous proposons d'utiliser notre exerciceur UpyLaB pour tester de manière autonome plusieurs exercices qui demandent d'utiliser l'instruction `if`.

Vous pouvez désormais réaliser les exercices UpyLaB 3.1 à 3.8 du Module 3, donnés aux pages qui suivent. Avant cela, lisez bien les rappels et recommandations ci-dessous.

RAPPELS IMPORTANTS

Note : Nous n'insisterons jamais assez sur le fait que, lors du développement de chacun de vos codes, il est plus que vivement conseillé d'utiliser un environnement de développement de type PyCharm ou Python Tutor pour *tester* ou même *déboguer* pas par pas votre code avant de passer aux tests *UpyLaB*. En effet, faire tester directement par UpyLaB un code dont vous n'avez pas une certaine confiance dans le fait qu'il soit correct s'avère souvent être un gouffre au niveau du temps étant donné qu'UpyLaB est moins précis qu'un environnement de développement pour vous expliciter où se situent les éventuels problèmes qui ne manqueront pas d'apparaître.

La démarche la plus efficace pour développer du code en Python est donc :

1. de comprendre exactement ce qui vous est demandé,
2. de développer une solution dans PyCharm en la testant, éventuellement avec Python Tutor, pour vous assurer qu'elle résout correctement le problème posé,
3. de tester l'ensemble de votre code dans UpyLaB (en le transférant dans UpyLaB grâce au copier / coller du code dans PyCharm) pour faire valider votre script.

Attention : Rappelez-vous également de bien suivre les autres **Conseils et consignes lors de la réalisation d'exercices UpyLaB** (cf section 2.5) et que le but de réaliser les exercices UpyLaB est de vous rendre autonome en programmation. C'est vous qui devez trouver la ou une solution, en terme de code Python, à chaque exercice demandé. Le forum ne doit donc pas être l'endroit où des codes complets ou substantiels sont échangés. Vos éventuelles questions doivent être précises et les réponses ponctuelles.

N'oubliez pas ce qui a été dit en début de cours sur les exercices UpyLaB !

- 1) Contrairement aux quiz, les exercices UpyLaB peuvent être testés autant de fois que vous le désirez.
- 2) Pour enregistrer votre réponse dans un exercice UpyLaB vous devez cliquer sur le bouton « Vérifier » même si vous savez que votre code n'est pas correct ; ce n'est pas un souci puisque le nombre de fois que vous vérifiez chaque exercice UpyLaB n'est pas limité.
- 3) Attention, si pour un exercice UpyLaB, la dernière vérification est négative, la plateforme ne retient pas si une vérification précédente avait été positive. En clair, la plateforme ne vous accorde les points que si la dernière vérification de cet exercice est positive.

EXERCICE UPYLAB 3.1 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui lit 3 nombres entiers, et qui, si au moins deux d'entre eux ont la même valeur, imprime cette valeur (le programme n'imprime rien dans le cas contraire).

Exemple 1

Avec les données lues suivantes :

```
2
1
2
```

le résultat à imprimer vaudra :

```
2
```

Exemple 2

Avec les données lues suivantes :

```
1
2
3
```

le programme n'affichera rien.

Exemple 3

Avec les données lues suivantes :

```
42
42
42
```

le résultat à imprimer vaudra :

```
42
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.1*.

3.3.4 Exercice UpyLaB 3.2 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui, si `temperature` (entier lu sur input correspondant à la température maximale prévue pour aujourd'hui) est strictement supérieur à 0, teste si `temperature` est inférieur ou égal à 10, auquel cas il imprime le texte :

— Il va faire frais

et qui, si `temperature` n'est pas supérieur à 0, imprime le texte :

— Il va faire froid

Dans les autres cas, le programme n'imprime rien.

Exemple 1

Avec la donnée lue suivante :

1

le résultat à imprimer vaudra :

Il va faire frais

Exemple 2

Avec la donnée lue suivante :

20

le programme n'affichera rien.

Exemple 3

Avec la donnée lue suivante :

-1

le résultat à imprimer vaudra :

Il va faire froid

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Faites attention d'écrire les messages à l'identique de ce qui est demandé (majuscule au début de la ligne, une espace entre chaque mot, etc) . Un conseil pour avoir des messages identiques est de les copier depuis l'énoncé pour les coller dans votre code.
- Lors de l'affichage des résultats, en cas d'erreur dans certains tests, UpyLaB pourra marquer : « Le résultat attendu était : aucun résultat ». Cela voudra bien dire qu'il ne faut rien imprimer dans ce cas.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.2*.

3.3.5 Exercice UpyLaB 3.3 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui lit trois entiers a , b et c en input. Ensuite :

- si l'entier c est égal à 1, alors le programme affiche la valeur de $a + b$;
- si c vaut 2, alors le programme affiche la valeur de $a - b$;
- si c est égal à 3, alors l'output sera la valeur de $a.b$ (produit de a par b);
- enfin, si la valeur 4 est assignée à la variable c , alors le programme affiche la valeur de $a^2 + a.b$;
- et si c contient une autre valeur, le programme affiche le message "Erreur".

Exemple 1

Avec les données lues suivantes :

```
3
2
1
```

le résultat à imprimer vaudra :

```
5
```

Exemple 2

Avec les données lues suivantes :

```
3
2
4
```

le résultat à imprimer vaudra :

```
15
```

Exemple 3

Avec les données lues suivantes :

```
3
2
5
```

le résultat à imprimer vaudra :

Erreur

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu en respectant l'orthographe, les majuscules / minuscules, les espacements, etc.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat : ", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.3*.

3.3.6 Exercice UpyLaB 3.4 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui teste la parité d'un nombre entier lu sur `input` et imprime `True` si le nombre est pair, `False` dans le cas contraire.

Exemple 1

Avec la donnée lue suivante :

13

le résultat à imprimer vaudra :

False

Exemple 2

Avec la donnée lue suivante :

42

le résultat à imprimer vaudra :

True

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester la parité d'un nombre, on peut s'intéresser au reste de la division de ce nombre par 2 et utiliser l'opérateur modulo.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.4*.

3.3.7 Exercice UpyLaB 3.5 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui lit en entrée deux nombres entiers strictement positifs, et qui vérifie qu'aucun des deux n'est un diviseur de l'autre.

Si tel est bien le cas, le programme imprime `True`. Sinon, il imprime `False`.

Exemple 1

Avec les données lues suivantes :

```
6
42
```

le résultat à imprimer vaudra :

```
False
```

Exemple 2

Avec les données lues suivantes :

```
5
42
```

le résultat à imprimer vaudra :

```
True
```

Consignes

- Notez qu'il n'est pas demandé de vérifier que les deux nombres en entrée sont strictement positifs. Vous pouvez supposer que ce sera le cas pour toutes les entrées proposées par UpyLaB lors des tests.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Dire qu'un nombre entier x est un diviseur d'un nombre entier y revient à dire que le reste de la division euclidienne de y par x est égal à 0.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.5*.

3.3.8 Exercice UpyLaB 3.6 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui imprime la **moyenne géométrique** $\sqrt{a \cdot b}$ (la racine carrée du produit de a par b) de deux nombres positifs a et b de type float lus en entrée.

Si au moins un de ces nombres est strictement négatif, le programme imprime le texte « Erreur ».

Exemple 1

Avec les données lues suivantes :

```
1.0
2.0
```

le résultat à imprimer vaudra approximativement :

```
1.4142135623730951
```

Exemple 2

Avec les données lues suivantes :

```
-1.0
2.0
```

le résultat à imprimer vaudra :

```
Erreur
```


Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(float(input()))` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour calculer la racine carrée d'un nombre positif, on pourra trouver une fonction utile dans le module `math`, ou se rappeler que cela correspond à la puissance d'exposant $\frac{1}{2}$.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm ou Python Tutor avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.6*.

3.3.9 Exercice UpyLaB 3.7 (Parcours Vert, Bleu et Rouge)

Énoncé

Dans mon casino, ma roulette comporte 13 numéros de 0 à 12 comme montrés ci-dessous :



Fig. 3.1 – Roulette

Le joueur a plusieurs types de paris possibles :

- il peut choisir de parier sur le numéro sortant, et dans ce cas, s'il gagne, il remporte douze fois sa mise ;
- il peut choisir de parier sur la parité du numéro sortant (pair ou impair), et dans ce cas, s'il gagne, il remporte deux fois sa mise ;
- enfin, il peut choisir de parier sur la couleur du numéro sortant (rouge ou noir), et dans ce cas aussi, s'il gagne, il remporte deux fois sa mise.

Si le joueur perd son pari, il ne récupère pas sa mise.

Pour simplifier, on suppose que le numéro 0 n'est ni rouge ni noir, mais est pair. Pour simplifier encore, on suppose que le joueur mise systématiquement 10 euros.

Écrire un programme qui aide le croupier à déterminer la somme que le casino doit donner au joueur.

Le programme lira, dans l'ordre, deux nombres entiers en entrée : le pari du joueur (représenté par un nombre entre 0 et 16, voir description plus bas), et le numéro issu du tirage (nombre entre 0 et 12). Le programme affichera alors le montant gagné par le joueur.

Entrées pour le pari du joueur :

- nombre entre 0 et 12 : le joueur parie sur le numéro correspondant
- 13 : le joueur parie sur pair
- 14 : le joueur parie sur impair
- 15 : le joueur parie sur la couleur rouge
- 16 : le joueur parie sur la couleur noire.

Exemple 1

Avec les données lues suivantes :

```
7
9
```

qui indiquent que le joueur parie sur le numéro 7 et que le numéro sorti est le 9,
le résultat à imprimer vaudra donc

```
0
```

Exemple 2

Avec les données lues suivantes :

```
16
4
```

qui indiquent que le joueur parie sur la couleur noire et que le numéro sorti est le 4 (qui est noir),
le résultat à imprimer vaudra donc

```
20
```

soit deux fois sa mise de 10 euros.

Exemple 3

Avec les données lues suivantes :

```
7
7
```

qui indiquent que le joueur parie sur le numéro 7 et que le numéro sorti est bien le 7,
le résultat à imprimer vaudra donc

```
120
```

soit douze fois sa mise de 10 euros.

Consignes

- Ne mettez pas d'argument dans les `input` : `data = input()` et non `data = input("Donnée suivante :")` par exemple.
- Le résultat doit juste faire l'objet d'un `print(res)` sans texte supplémentaire (pas de `print("résultat = ", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :






Conseils

- Lisez bien l'énoncé. Que représente chacune des deux entrées ?
- Même si le hasard intervient dans cette situation, ici il ne faut pas utiliser le module `random`. Nous ne simulons pas le tirage de la roulette, c'est l'utilisateur (le croupier) qui saisira les deux entrées correspondant au pari du joueur, et au résultat du tirage.
- Pensez à utiliser les opérateurs logiques, mais faites attention à leur priorité : voir section 3.2.3.
- Si rien ne marche : consultez la [FAQ sur UpyLaB 3.7](#).

3.3.10 Exercice UpyLaB 3.8 (Parcours Rouge)

Énoncé

Les cinq polyèdres réguliers de Platon sont représentés ci-dessous, avec la formule de leur volume.

Nom	Volume	Image
Tétraèdre	$\frac{\sqrt{2}}{12} a^3$	
Cube	a^3	
Octaèdre	$\frac{\sqrt{2}}{3} a^3$	
Dodécaèdre	$\frac{15+7\sqrt{5}}{4} a^3$	
Icosaèdre	$\frac{5(3+\sqrt{5})}{12} a^3$	

Écrire un programme qui lit :

- la première lettre en majuscule du nom du polyèdre ("T", "C", "O", "D" ou "I"),
- la longueur de l'arête du polyèdre,

et qui imprime le volume du polyèdre correspondant.

Si la lettre lue ne fait pas partie des cinq initiales, le programme imprime le message "Polyèdre non connu".

Exemple 1

Avec les données lues suivantes :

```
C
2.0
```

le résultat à imprimer vaudra :

```
8.0
```

Exemple 2

Avec les données lues suivantes :

```
I
2.0
```

le résultat à imprimer vaudra approximativement :

```
17.4535599249993
```

Exemple 3

Avec les données lues suivantes :

```
A
2.0
```

le résultat à imprimer vaudra :

```
Polyèdre non connu
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(float(input()))` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- La fonction `input` retourne systématiquement un résultat de type chaîne de caractères. Si ce qui est à lire est un texte ou un simple caractère, on peut donc utiliser directement le retour de cette fonction, mais s'il s'agit d'un nombre, il faut penser à le convertir à l'aide des fonctions `int` ou `float`.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.8*.

3.4 Les instructions répétitives `while` et `for`

3.4.1 L'instruction `while`

INTRODUISONS L'INSTRUCTION `WHILE`

La présente section vous explique tout ce qui vous sera nécessaire sur l'instruction `while`. Cette vidéo introduit l'instruction `while` qui permet d'exécuter certaines instructions de façon répétitive tant qu'une certaine condition est vérifiée. Comme le traitement peut s'exécuter plusieurs fois, nous parlerons pour l'instruction `while` mais aussi pour l'instruction `for` que nous verrons juste après, d'instructions répétitives `while` et `for`. Encore une fois, nous partons d'un exemple simple pour expliquer le fonctionnement de l'instruction `while`, et ensuite nous élaborons pour montrer différentes possibilités.

Notons aussi que dans le jargon informatique, les instructions répétitives `while` et `for` sont également appelées « boucles » `while` ou `for` pour marquer cette notion de répétition.

INTRODUCTION DE L'INSTRUCTION RÉPÉTITIVE `WHILE`

Note : Voir la vidéo de la section 3.4.1 : L'instruction `while`

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction répétitive `while`.

Script réalisé dans la vidéo

```
DISTANCE = 3844.0e5
nombre_pliages = 0
epaisseur = 0.0001
while epaisseur < DISTANCE :
    epaisseur = 2 * epaisseur
    nombre_pliages = nombre_pliages + 1
print('nombre de pliages nécessaire : ', nombre_pliages)
```

La vidéo en bref

La vidéo introduit l'instruction répétitive `while` avec un petit exemple.

La syntaxe d'une instruction `while` est

```
while condition :
    instructions
```

avec le mot-clé `while`, suivi d'une expression booléenne que l'on appelle souvent condition de continuation, suivie de « : », puis de toutes les instructions qui sont indentées par rapport à cette première ligne et qui constituent les instructions associées au `while`.

Dans le jargon informatique, on appelle ces instructions le « corps de la boucle `while` ».

Lors de l'exécution d'une instruction `while`, la condition de continuation est évaluée, et si elle est vraie, le corps de la boucle `while` est exécuté, ensuite, on recommence cette séquence : évaluation de la condition, exécution du corps de la boucle jusqu'au moment où la condition est évaluée à faux.

Notons que cette condition peut être directement fausse ; dans ce cas, le corps de la boucle n'est pas exécuté et l'interpréteur passe directement à l'instruction après l'instruction `while`.

C'EST FACILE D'ALLER SUR LA LUNE

42 correspond à la première valeur de x telle que $0.0001.2^x \geq 3.844.10^8$. Notons pour les forts en math que cela correspond à la première solution entière de l'inéquation $384400000 < 0.0001.2^n$ c'est-à-dire à la première valeur entière supérieure à $\log_2(384400000000)$ (en python `ceil(log(384000000000, 2))` où `log` et `ceil` (valeur plafond) ont été importées du module `math` (`from math import ceil, log`). C'est donc très facile d'aller sur la Lune : il suffit de plier une feuille 42 fois et de se mettre dessus !

3.4.2 Syntaxe du `while` et calcul du plus grand commun diviseur

SYNTAXE DE L'INSTRUCTION RÉPÉTITIVE `WHILE`

La syntaxe de l'instruction répétitive `while` est :

```
while condition:
    instructions
```

où `condition` est une expression booléenne appelée condition de continuation et `instructions` est une instruction ou une séquence d'instructions indentées par rapport à la ligne `while condition:`, que l'on nomme généralement corps de la boucle `while`.

CALCUL DU PLUS GRAND COMMUN DIVISEUR

Un autre exemple plus mathématique mais simple, qui illustre parfaitement l'utilisation de l'instruction répétitive `while` est le calcul du plus grand commun diviseur de deux nombres entiers positifs. Rappelons d'abord ce qu'est le plus grand commun diviseur.

Définition du plus grand commun diviseur (pgcd)

Ayant deux nombres entiers positifs x et y (par exemple 132 et 36),

le *plus grand diviseur* (pgcd) des deux nombres est le nombre d qui est :

- **un diviseur entier à la fois de x et de y , c'est-à-dire que**
 - x divisé par d donne un nombre entier,
 - et de même pour y divisé par d ,
- **et de plus tel qu'il n'existe pas d'autre entier strictement plus grand que d également diviseur entier à la fois de x et de y .**

Ainsi on peut vérifier que 12 est le plus grand commun diviseur de 132 et 36.

Calcul du plus grand commun diviseur (pgcd) avec la méthode d'Euclide améliorée

Le calcul du plus grand commun diviseur est un très bel exemple d'utilisation de l'instruction `while` vu sa simplicité. La mini capsule vidéo suivante explique le problème du calcul du pgcd(x , y) et sa solution simple en Python.

MÉTHODE DE CALCUL DU PGCD(X,Y)

Note : Voir la vidéo de la section 3.4.2 : Calcul du PGCD

CONTENU DE LA VIDÉO

La vidéo précédente présente un script qui calcule le plus grand commun diviseur de deux nombres lus en entrée.

Script réalisé dans la vidéo

```
x = int(input("x = "))
y = int(input("y = "))
while y > 0:
    x, y = y, x % y
    print(x, y)
print("pgcd = ", x)
```

La vidéo en bref

Le script met directement en application deux propriétés mathématiques du pgcd, soit :

- Si x est positif et y strictement positif : $\text{pgcd}(x, y)$ est équivalent à $\text{pgcd}(y, x \% y)$
 Par exemple $132 \% 36$ vaut 24 et donc $\text{pgcd}(132, 36)$ vaut $\text{pgcd}(36, 24)$
 ce qui est intéressant car y et $x \% y$ seront plus petits que x et y .
- $\text{pgcd}(x, 0)$ vaut x .

FONCTION GCD DU MODULE MATH

Notons que Python met à votre disposition la fonction `gcd` du module `math`, qui calcule le plus grand commun diviseur de deux nombres entiers.

3.4.3 Conjecture de Syracuse

CONJECTURE DE SYRACUSE

Mettons en pratique ce que nous venons de voir sur l'instruction `while`.

Aidons les mathématiciens à valider une conjecture pourtant simple qui leur donne du fil à retordre. Le texte sur la conjecture qui suit est repris de l'article Wikipedia sur la conjecture de Syracuse.

D'abord par un peu d'histoire :

En 1952, lors d'une visite à Hambourg, le mathématicien allemand Lothar Collatz expliqua le « problème $3x+1$ » à Helmut Hasse. Ce dernier le diffusa en Amérique à l'Université de Syracuse sous le nom de « suite de Collatz » ou de « suite de Syracuse ». Entre temps, le mathématicien polonais Stanislas Ulam le répand dans le Laboratoire national de Los Alamos. Dans les années 1960, le problème est repris par le mathématicien Shizuo Kakutani qui le diffuse dans les universités Yale et Chicago.

Définissons le problème :

La suite de Collatz pour un nombre entier n strictement positif est définie comme suit :

- la première valeur de la suite est le nombre n lui-même,
- si n est pair la valeur suivante sera n divisé par 2,
- sinon la valeur suivante est $3n + 1$ (n multiplié par 3 auquel on rajoute 1).

En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur.

Par exemple, à partir de 14, on construit la suite des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2...

C'est ce qu'on appelle la suite de Collatz ou suite de Syracuse du nombre 14.

Après que le nombre 1 a été atteint, la suite des valeurs (1,4,2,1,4,2...) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial.

Si l'on était parti d'un autre entier, en lui appliquant les mêmes règles, on aurait obtenu une suite de nombres différente. A priori, il serait possible que la suite de Syracuse de certaines valeurs de départ n'atteigne jamais la valeur 1, soit qu'elle aboutisse à un cycle différent du cycle trivial, soit qu'elle diverge vers l'infini. Or, on n'a jamais trouvé d'exemple de suite obtenue suivant les règles données qui n'aboutisse pas à 1 et, par suite, au cycle trivial.

La conjecture de Syracuse est l'hypothèse mathématique selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Cette conjecture mobilisa tant les mathématiciens durant les années 1960, en pleine guerre froide, qu'une plaisanterie courut selon laquelle ce problème faisait partie d'un complot soviétique visant à ralentir la recherche américaine.

Un grand mathématicien du 20ème siècle, Paul Erdős a dit, à propos de la conjecture de Syracuse : « Les mathématiques ne sont pas encore prêtes pour de tels problèmes ».

IMPLÉMENTATION DE LA CONJECTURE DE SYRACUSE

Nous voulons écrire un code qui génère la suite de Syracuse d'un nombre n lu sur input, jusqu'à ce qu'elle atteigne la valeur 1.

Développons le problème.

Le nombre n est lu sur input :

```
n = int(input('valeur du nombre n dont on veut tester la conjecture'))
```

Calculer la valeur suivante peut se faire avec une instruction `if`

```
if n % 2 == 0 : # si un nombre entier modulo 2 vaut 0, il est pair
    n = n // 2
else:          # cas où le nombre est impair
    n = 3 * n + 1
```

Enfin pour tester si la suite satisfait la conjecture, il faut continuer à calculer de nouvelles valeurs jusqu'à ce que n vaille 1 :

```
while n != 1:
    if n % 2 == 0 : # si un nombre entier modulo 2 vaut 0, il est pair
        n = n // 2
    else:          # cas où le nombre est impair
        n = 3 * n + 1
```

Notons que si la conjecture n'est pas satisfaite, ce programme boucle indéfiniment, ce qui signifie que ici que la boucle `while` ne s'arrêtera jamais sauf si l'utilisateur coupe l'exécution du programme. Des informaticiens ont testé la conjecture pour tous les entiers jusqu'à des valeurs de l'ordre de 2^{60} sans trouver de contre-exemple. Mais à l'heure actuelle aucune preuve mathématique n'existe qu'il n'en existe pas un plus grand qui ne la satisfait pas.

N'hésitez pas à mettre tous les morceaux de code ensemble pour obtenir un programme complet. Pour voir si votre code complet ressemble au nôtre, une solution vous est proposée ici.

Une solution pour Syracuse

```
n = int(input('entier strictement positif : '))
while n != 1:
    print(n)
    if n % 2 == 0:
```

(suite sur la page suivante)

(suite de la page précédente)

```

        n = n // 2
    else:
        n = n * 3 + 1
print(n) # imprime 1

```

3.4.4 Deux canevas classiques rencontrés avec une boucle while

TRAITEMENT JUSQU'À CE QU'UNE CERTAINE CONDITION SOIT VÉRIFIÉE

Assez souvent le programmeur qui a un code à produire retrouve des « canevas » classiques qui lui permettent rapidement de savoir quelles instructions utiliser. Le canevas type pour une instruction `while` contient souvent les quatre parties *initialisation*, *condition*, *traitement*, *changement* :

```

initialisation
while condition:
    traitement
    changement

```

avec :

- une partie *initialisation* avant le `while` permettant après de tester la condition de continuation;
- la ligne contenant le `while` avec la condition de continuation qui détermine si une itération supplémentaire doit être réalisée dans le `while`;
- la partie *traitement* dans le corps du `while`;
- une partie que l'on peut qualifier de *changement* à la fin du corps du `while` qui constitue une sorte de *réinitialisation* qui va permettre le test suivant de la condition de continuation.

Par exemple, dans le code *Syracuse* que nous venons de présenter :

Les parties *initialisation* et *changement* initialisent et modifient `n` qui sera testé au niveau de la condition de `while`; la partie *traitement* imprime la nouvelle valeur de `n`.

```

n = "première valeur"
while "n ne correspond pas à la valeur pour arrêter":
    traitement
    n = "nouvelle valeur"

```

LECTURE D'UNE SUITE DE DONNÉES

Il est fréquent que l'on lise une suite de données et fasse un certain traitement sur ces données tant que le code ne lit pas une valeur qui spécifie la fin de la suite des données à lire.

Par exemple si l'on doit lire une suite de données jusqu'à une valeur dite « sentinelle » valant "F", le canevas classique, avec deux lignes d'input nécessaires, est le suivant :

```

x = input('Première donnée : ')
while x != 'F':
    traitement
    x = input('Donnée suivante : ')

```

N'oubliez pas que la fonction `input()` renvoie un résultat de type chaîne de caractères (par exemple '1', '32', '-66' ou 'F') et que la fonction `int(x)` ou `float(x)` transforme une chaîne de caractères `x`, qui dénote respectivement un nombre entier, et fractionnaire, en la valeur correspondante de type `int` ou `float`. Vous pouvez bien sûr utiliser les deux fonctions en séquence directe, par exemple :

```

n = int(input())

```

ou non, par exemple :

```
x = input()
if x != 'F':
    n = int(x)
    ...
...
```

Certains des exercices UpyLaB qui suivent utiliseront ces canevas.

3.4.5 Des programmes qui bouclent indéfiniment

PROGRAMME QUI BOUCLE

Une des hantises du programmeur est un programme qui *boucle*, c'est-à-dire qui réexécute sans fin et de façon erronée la même séquence d'instructions. L'utilisation de l'instruction `while` est une des causes possibles d'un tel programme qui boucle indéfiniment, comme illustré dans la capsule vidéo suivante :

EXEMPLES DE PROGRAMMES QUI BOUCLENT INDÉFINIMENT

Note : Voir la vidéo de la section 3.4.5 : Programmes qui bouclent

CONTENU DE LA VIDÉO

La vidéo précédente présente des exemples d'instructions répétitives `while` qui bouclent indéfiniment.

Scripts réalisés dans la vidéo

```
i = 0
while i >= 0 :
    i = i + 1
    print(i)
```

```
i = 0
while i >= 0 :
    i = i + 1
print("le programme n'arrivera jamais ici")
```

La vidéo en bref

Des codes peuvent ne pas s'arrêter tout seuls, comme montré avec les scripts donnés dans la vidéo, où des instructions `while` ont la condition de continuation qui n'est jamais évaluée à faux durant leur exécution. Dans ce cas, il faudra arrêter l'exécution de ces codes généralement erronés, grâce à une touche clavier ou un bouton spécifique d'interruption d'exécution.

3.4.6 L'instruction for

INSTRUCTION FOR

La seconde instruction répétitive de Python est l'instruction `for`, également appelée boucle `for`.

La vidéo suivante illustre le fonctionnement de l'instruction `for` par un exemple très simple. Cet exemple introduit également la notion de séquence sur laquelle nous reviendrons plus en détails dans le module de cours suivant. Nous allons voir que l'instruction `for` répète un traitement pour tous les éléments d'une séquence.

INTRODUCTION DE L'INSTRUCTION RÉPÉTITIVE FOR

Note : Voir la vidéo de la section 3.4.6 : L'instruction `for`

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction répétitive `for`.

Scripts réalisés dans la vidéo

```
for c in "Bonjour":  
    print(c)
```

```
for i in range(5):  
    print(i)
```

```
somme = 0  
for val in range(5):  
    somme = somme + val  
print(somme)
```

La vidéo en bref

L'instruction `for` réalise un traitement donné pour tous les éléments d'une séquence.

La vidéo montre trois scripts : un premier qui réalise un traitement sur tous les caractères d'une chaîne de caractères, et deux autres qui utilisent la fonction `range` pour faire un traitement avec une séquence de valeurs entières.

La syntaxe de base d'une instruction `for` est ensuite donnée dans la vidéo (voir plus loin dans le cours).

RANGE

Dans la vidéo précédente nous avons vu que le code :

```
for i in range(5):  
    print(i)
```

affiche

```
0  
1  
2  
3  
4
```

Notez donc que le code affiche 5 valeurs de 0 à 4 (donc 5 non compris).

Nous verrons en section 3.5.11 que `range` peut également être utilisé avec 2 ou 3 arguments ce qui étend ses possibilités.

3.4.7 Syntaxe du for, carrés, étoiles et autres polygones réguliers

SYNTAXE DE L'INSTRUCTION FOR

Nous avons vu dans la vidéo précédente que la syntaxe de l'instruction répétitive `for` est :

```
for c in sequence:
    instructions
```

où :

- `c` est la *variable de contrôle* qui reçoit séquentiellement chacune des valeurs de la séquence
- `sequence` est, ou génère, la séquence de valeurs utilisée
- `instructions` constitue le *corps de la boucle* `for`

Pour que votre code fonctionne correctement, il faut absolument que les `instructions` dans le *corps du for* ne modifient pas la variable `c` (et ce même si Python ne l'interdit pas explicitement). Si vous modifiez `c`, votre code va souvent produire des effets assez aléatoires !

DESSIN D'UN CARRÉ, D'UN POLYGONE RÉGULIER OU D'UNE ÉTOILE AVEC TURTLE

Pour mettre en pratique l'instruction `for`, utilisons à nouveau le module `turtle` pour dessiner des polygones.

Une boucle `for` permet de dessiner très simplement un carré (par exemple dont les côtés ont une longueur de 100).

```
import turtle
for i in range(4):
    turtle.forward(100)
    turtle.left(90)
```

À vous ! Reprenez ce code pour écrire et exécuter dans PyCharm un script qui trace un polygone régulier à n côtés avec n au moins égal à 3, ou une étoile à n branches avec n au moins égal à 5 et de valeur impaire.

Rappelons-nous (module 2 Activité 2.6.3) que pour dessiner un polygone à n côtés (par exemple $n = 5$), l'angle intérieur entre deux côtés est de $360^\circ/n$. Si n vaut 5 cela donne donc 72° .

Rappelons-nous également que si l'on veut dessiner des étoiles à n branches et en supposant n impair au moins égal à 5, l'angle intérieur sera de $(n - 1) * 180^\circ/n$ (par exemple 144° pour n valant 5).

POUR LES AMOUREUX DES ÉTOILES OU DES MATHÉMATIQUES

Dans le paragraphe précédent nous avons expliqué comment dessiner une étoile ayant un nombre impair, au moins égal à 5, de branches. En supposant que l'on désire dessiner une étoile à n branches par une succession de segments contigus, une telle étoile peut être dessinée pour toutes les valeurs de n impaires à partir de 5 et la plupart des valeurs paires.

En dessous de 5 nous ne pouvons tracer d'étoile. Une exception notoire au dessus de 5 est l'étoile à 6 branches qui, comme le montre la figure « Etoile à 6 branches », demande d'effectuer deux tracés distincts où à chaque fois, on dessine un triangle.

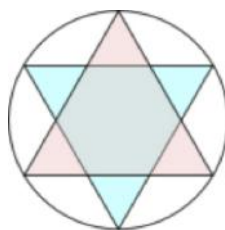


Fig. 3.2 – Etoile à 6 branches

Confectionnons à présent un code qui reçoit en entrée le nombre n de branches et qui, s'il est possible de tracer une étoile à n branches en un seul tracé contigu, la dessine avec `turtle`.

Pour trouver comment faire, analysons la situation avec une valeur n paire. Prenons l'exemple de n valant 8. Le principe est de prendre un cercle et de le découper en n sections comme on découpe une tarte en n parts égales. Le dessin ci-dessous représente une telle découpe pour n valant 8.

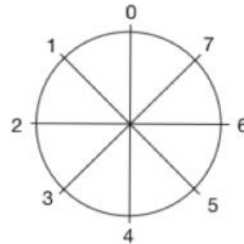


Fig. 3.3 – Découpe du cercle en 8 parts égales.

Ensuite, le principe est de partir du point 0 et de tracer un segment vers un autre point. Nous répétons cela n fois. Partant de 0, le point suivant peut par exemple être le point 3. Ensuite, l'idée est de continuer avec le même incrément, donc le point 6 et ainsi de suite, en ajoutant 3 à chaque fois. Quand on arrive à un nombre plus grand que 7, on effectue un modulo 8. Dans notre exemple, après 6 on aura $6 + 3$ donne 9 ; dans ce cas 9 modulo 8 nous dit que le point suivant est 1.

Donc le tracé passe séquentiellement par les points 0, 3, 6, 1, 4, 7, 2, 5, et retour à 0, comme le montre le schéma suivant.

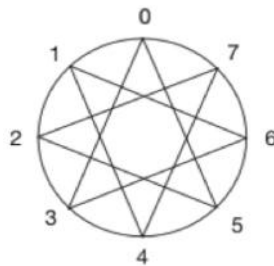


Fig. 3.4 – Étoile à 8 branches

De façon générale avec n sommets et n au moins égal à 5 (en dessous on ne pourra tracer une étoile), partant du point 0, il faut trouver un incrément inc qui modulo n fera passer par tous les points avant de revenir au point 0. L'incrément inc devra être plus grand que 1 sinon, nous traçons un polygone et non pas une étoile. Nous prenons aussi inc inférieur à $n/2$ sinon le tracé se fera en traçant par la droite et non par la gauche. Il se peut que plusieurs incréments soient possibles. Si nous désirons l'étoile avec les branches les plus fines possibles, il faut que inc soit le plus proche de $n/2$. Quand n est impair la valeur inc telle que $n = 2 \cdot inc + 1$ fonctionne toujours (en Python il suffit si n est impair de définir $inc = n // 2$). Si n est pair ce n'est pas aussi facile. Mais les mathématiques nous disent qu'il faut que inc soit la plus grande valeur entière strictement plus petite à $n / 2$ telle que le $pgcd(n, inc)$ soit égal à 1.

Le code Python suivant effectue le travail :

```
""" trace une étoile à n côtés, si elle peut l'être sans lever la plume """

import turtle
from math import gcd # fonction du module math qui calcule le pgcd de 2 nombres

LONGUEUR = 100 # taille de chaque segment de l'étoile

n = int(input("Combien de branches désirez-vous ? :"))
inc = (n-1) // 2
while gcd(n, inc) > 1:
    inc = inc - 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
if inc == 1 :
    print("Impossible de dessiner une étoile à", n, "branches en un tenant")
else:
    angle = 180 - (n - 2 * inc) * 180 / n
    for i in range(n):
        turtle.forward(LONGUEUR)
        turtle.left(angle)
```

Notons qu'à partir de 5 branches, l'étoile à 6 branches est la seule qui ne puisse être tracée avec ce programme. En effet 6 est la seule valeur n entière positive telle qu'aucune valeur entière positive i dans l'intervalle entre 1 non compris et $n/2$ non compris, soit telle que $\text{pgcd}(i, n) == 1$.

Bons dessins d'étoiles !

3.4.8 Quiz sur while et for

QUIZ POUR RÉCAPITULER SUR LES INSTRUCTIONS WHILE ET FOR

Note : Voir le quiz de la section 3.4.8

3.5 Code avec while et for dans la pratique

Mettons ensemble en pratique ce que nous avons vu sur les boucles `while` et `for` Python. Avant de vous proposer de résoudre, de façon autonome, des exercices UpyLaB, commençons par des exercices avec correction pour comparer votre solution à la nôtre et vous guider dans votre apprentissage pratique : une activité sur la suite de Fibonacci suivie d'une activité de dessin d'un pavé hexagonal.

3.5.1 La suite de Fibonacci

SUITE DE FIBONACCI

Selon le site « [image des mathématiques](#) » du CNRS :

La suite de Fibonacci commence ainsi :

0 1 1 2 3 5 8 13 21 34 55 89 144 233 ...

Ses deux premiers termes sont 0 et 1, et ensuite, chaque terme successif est la somme des deux termes précédents. Ainsi

- $0 + 1 = 1$
- $1 + 1 = 2$
- $1 + 2 = 3$
- $2 + 3 = 5$
- $3 + 5 = 8$
- etc.

Son inventeur est Léonard de Pise (1175 - v.1250), aussi connu sous le nom de Leonardo Fibonacci. Introduite comme problème récréatif dans son fameux ouvrage *Liber Abaci*, la suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! En effet, elle y décrit la croissance d'une population de lapins sous des hypothèses très simplifiées, à savoir : chaque couple de lapins, dès son troisième mois d'existence, engendre chaque mois un nouveau couple de lapins, et ce indéfiniment.

Partons donc d'un couple de lapins le premier mois. Le deuxième mois, on n'a toujours que ce même couple, mais le troisième mois on a déjà 2 couples, puis 3 couples le quatrième mois, 5 couples le cinquième mois, etc. La croissance de cette population est bel et bien décrite par la suite de Fibonacci en partant de 0 et 1 comme deux premiers nombres de cette suite.

Nous vous proposons d'écrire deux petits programmes qui calculent et impriment les premiers termes de la suite de Fibonacci dans deux cas de figure :

- **problème 1** : pour les n premiers termes
- **problème 2** : pour tous les termes inférieurs à une valeur n donnée.

Pour simplifier, supposons que pour les 2 problèmes, n est supérieur ou égal à 2.

Ces deux problèmes illustrent parfaitement la règle suivante qui guide le choix, lorsque l'on doit répéter un traitement, de l'utilisation de l'instruction `for` ou `while` :

Avertissement : Règles : Lorsqu'un traitement répétitif doit être réalisé par un programme Python,

- **l'utilisation d'une instruction répétitive `for` est à favoriser**
quand nous savons facilement, au début de l'exécution de la boucle, combien d'itérations devront être réalisées ou si l'on demande de traiter tous les éléments d'une séquence comme nous le verrons au module sur les séquences ;
- **l'utilisation d'une instruction répétitive `while` est à favoriser**
quand a priori le nombre d'itérations n'est pas facilement déterminable avant le début de son exécution.
- N'essayez pas cette expérience avec votre couple de lapins !

Les deux codes commencent donc par « lire une valeur entière n » (c'est-à-dire, lire une valeur entière et assigner cette valeur à la variable `n`).

Comme on ne sait pas calculer n termes en une simple séquence d'assignations et de `if` puisque que n n'est connu qu'à l'exécution quand le script du code est déjà écrit, il est clair que pour les deux codes, il faut utiliser une instruction répétitive (une boucle). Une façon habituelle de coder dans ce type d'exemple est de calculer un nouveau terme à chaque itération.

Encore faut-il savoir comment mettre cela en musique et si c'est une instruction `while` ou `for` qu'il faut utiliser.

Pour les deux problèmes qui nous préoccupent, comme pour le calcul du plus grand commun diviseur, une façon classique de calculer un terme suivant est d'avoir deux variables, que l'on nomme par exemple `prec` et `succ`, qui vont contenir l'avant-dernier et le dernier terme calculés.

Initialement on a

```
prec = 0
succ = 1
```

À chaque itération, il faut calculer le terme suivant mais aussi conserver l'avant-dernier, c'est-à-dire celui qui était dans `succ` pour nous permettre de continuer.

Typiquement on pourra écrire :

```
prec, succ = succ, prec + succ
```

SUITE DE FIBONACCI : PROBLÈME 1

Si l'on veut écrire les n premiers termes de la suite, il suffit de les calculer un à un et de les afficher, par exemple avec

```
for i in range(n):
    prec, succ = succ, prec + succ
    print(succ)
```

Plus précisément, le code complet animé par Python Tutor du problème 1 est :

Code pour le problème 1

```
n = int(input('nombre de termes à calculer de la suite de Fibonacci : '))
prec = 0
succ = 1
print(prec, end = ' ')
print(succ, end = ' ')
for i in range(n-2):
    prec, succ = succ, prec + succ
    print(succ, end = ' ')
print()
```

Commentaires sur le code

Le `range(n-2)` est dû au fait que les 2 premiers termes ont déjà été imprimés et qu'il en reste donc $n-2$ à calculer.

Notons aussi l'explicitation de l'argument nommé `end` dans les `print`, pour imprimer les résultats sur la même ligne (avec `end = ' '`, chaque `print` est séparé par une espace), et le dernier `print()` permet de passer à la ligne après avoir imprimé la dernière valeur.

SUITE DE FIBONACCI : PROBLÈME 2

Pour le problème 2, on ne sait pas a priori combien d'itérations devront être effectuées. Dans ce cas, un `while` est la bonne instruction à utiliser.

Le code animé par Python Tutor du problème 2 est donc :

Code pour le problème 2

```
n = int(input('borne supérieur à ne pas dépasser pour calculer la suite de Fibonacci : '))
prec = 0
succ = 1
print(prec, end = ' ')
while succ < n :
    print(succ, end = ' ')
    prec, succ = succ, prec + succ
print()
```

Commentaires sur le code

Notons qu'ici en plus d'utiliser un `while`, `succ` est imprimé juste après le test (condition de continuation du `while`) pour s'assurer qu'il faut effectivement l'imprimer.

Le programmeur devra toujours essayer d'avoir non seulement un code correct pour toutes les exécutions possibles, mais également le plus efficace possible tout en étant le plus clair possible. Nous reviendrons sur cet aspect plus tard.

Note : une instruction `for` peut en général facilement être traduite par un `while` même si cela donne du code plus compliqué et donc moins lisible ; c'est donc à déconseiller. Par exemple :

AUTRE EXEMPLE DE CODE QUI UTILISE UNE INSTRUCTION FOR

Une instruction `for` peut être réécrite en un `while` comme le montre l'exemple ci-dessous :

```
for i in range(10) :
    print(i)
```

peut être écrit comme suit :


```
i = 0
while i < 10 :
    print(i)
    i = i + 1
```

Note : Codes animés avec Python Tutor : voir cours en ligne en section 3.5.1

Pour être précis, les deux codes n'ont pas un effet totalement équivalent : à la sortie du `for`, `i` vaut 9 et pas 10 comme à la sortie du `while`.

3.5.2 Un pavé du projet Vasarely avec des `for`

CODE D'UN PAVÉ HEXAGONAL AVEC DES FOR IMBRIQUÉS

Un exemple beaucoup plus difficile à ce stade de votre apprentissage est d'écrire un code qui utilise `turtle` pour faire un pavé hexagonal tel que demandé au module précédent, mais en ayant le code le plus succinct possible grâce à l'utilisation d'instructions `for`.

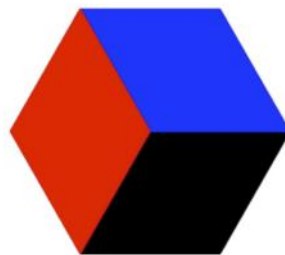


Fig. 3.5 – Pavé coloré

En effet, on peut constater qu'un tel pavé est constitué de trois losanges, chacun constitué de quatre côtés de même taille, mais reliés avec des angles alternativement de 60° ou de 120° .

Comme nous n'avons pas encore vu la notion de séquence qui nous aidera pour faire l'exemple complet, essayez d'écrire le code qui utilise en particulier des instructions `for` pour tracer l'ensemble des lignes de l'hexagone, mais sans changer de couleur ni remplir les surfaces, tel que montré ici :

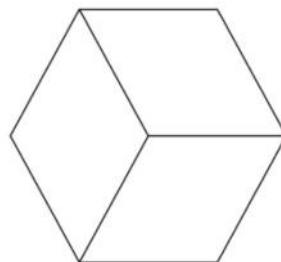


Fig. 3.6 – Pavé non coloré

Aide : on peut voir que le code suivant imprime quatre fois la valeur d'une variable `angle` en alternant entre les deux valeurs 120 et 60 (120, 60, 120, 60). En effet ce code utilise le fait que $180 - 120$ vaut 60 et $180 - 60$ vaut 120.

```
angle = 120
for j in range(4):
    print(angle)
    angle = 180 - angle
```

Par ailleurs,

- le premier losange commence son tracé par un segment de ligne horizontal,
- le second, par un segment qui forme un angle de 120° par rapport au premier segment,
- le troisième, par un segment qui forme un angle de 120° par rapport au second segment.

En résumé, il faut répéter trois fois :

- tracer un losange (où à chaque fois il faut tracer quatre segments)
- ensuite tourner à droite de 120° .

Le canevas général sera donc :

```
for i in range(3) : # à chaque itération, trace un losange
    for j in range(4) : # à chaque itération, trace un segment
        ...
```

À vous de jouer pour écrire un programme complet réalisant le tracé du pavé hexagonal.

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution ? Nous vous en proposons une :

```
""" auteur: Thierry Massart
    date : 9 avril 2018
    but du programme : trace avec turtle les contours d'un pavé hexagonal
"""
import turtle
for i in range(3): # à chaque itération, trace un losange
    angle = 120
    for j in range(4): # à chaque itération, trace un segment
        turtle.forward(100)
        turtle.left(angle)
        angle = 180 - angle
    turtle.right(120)
turtle.hideturtle()
```

3.5.3 Mise en pratique autonome : exercices UpyLaB 3.9 et suivants

MISE EN PRATIQUE AUTONOME AVEC UPYLAB

Maintenant que nous avons ensemble rédigé des scripts avec des « boucles », il vous reste l'ultime étape dans l'apprentissage de ce module : devenir autonome dans la rédaction de tels codes. Nous vous demandons de réaliser les exercices UpyLaB 3.9 à 3.17 du Module 3 proposés dans les pages qui suivent. En particulier, l'exercice UpyLaB 3.14 est une version plus élaborée du petit jeu de devinette présenté en section 3.3 du présent module.

EXERCICE UPYLAB 3.9 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui demande à l'utilisateur combien de plis de papier sont nécessaires pour se rendre sur la Lune, et pose la question tant que l'utilisateur n'a pas saisi la bonne réponse. Si la réponse saisie par l'utilisateur n'est pas correcte, le programme

affiche le message "Mauvaise réponse.", puis pose à nouveau la question. Si la réponse saisie par l'utilisateur est correcte, le programme affiche le message "Bravo !", et s'arrête. Exemple

Dans cet exemple d'exécution, le texte est affiché par le programme, alors que les nombres sont saisis par l'utilisateur :

```
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 666
Mauvaise réponse.
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 3
Mauvaise réponse.
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 42
Bravo !
```

Consignes

- UpyLaB va vérifier le bon fonctionnement du programme en comparant ce qui est affiché à l'écran avec ce qu'il attend. Veillez donc à bien respecter le texte à afficher (casse, espaces, ponctuation...).
- En particulier, pour lire les données utilisez précisément l'instruction :

```
int(input("Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : "))
```

et pour afficher utilisez précisément les instructions

```
print("Mauvaise réponse.")
```

et

```
print("Bravo !")
```

Conseils

Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.

3.5.4 Exercice UpyLaB 3.10 (parcours vert, bleu et rouge)

Énoncé

Écrire un programme qui calcule la taille moyenne (en nombre de salariés) des Petites et Moyennes Entreprises de la région.

Les tailles seront données en entrée, chacune sur sa propre ligne, et la fin des données sera signalée par la valeur sentinelle -1. Cette valeur n'est pas à comptabiliser pour le calcul de la moyenne, mais indique que l'ensemble des valeurs a été donné.

Après l'entrée de cette valeur sentinelle -1, le programme affiche la valeur de la **moyenne arithmétique** calculée.

On suppose que la suite des tailles contient toujours au moins un élément avant la valeur sentinelle -1, et que toutes ces valeurs sont positives ou nulles.

Exemple 1

Avec les données lues suivantes :

```
11
8
14
5
-1
```

le résultat à imprimer vaudra :

```
9.5
```

Exemple 2

Avec les données lues suivantes :

```
12
6
7
-1
```

le résultat à imprimer vaudra approximativement :

```
8.333333333333334
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il sera utile de créer deux variables, une qui stockera la somme des valeurs entrées et l'autre leur nombre, et que l'on actualisera après chaque lecture.
- N'oubliez pas de supprimer les textes à l'intérieur des appels à `input` lorsque vous soumettez le code à UpyLaB.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.10*.

3.5.5 Note sur la fonction print

PRINT

Nous avons vu que la fonction `print()` affiche l'ensemble des valeurs données en argument. Ainsi le code suivant :

```
date_1 = 1515
date_2 = 1789
print("Bataille de Marignan", date_1)
print("Révolution française", date_2)
```

affiche les deux lignes

```
Bataille de Marignan 1515
Révolution française 1789
```

Notez donc que :

- 1) chaque print se termine par un passage à la ligne suivante;
- 2) une espace sépare chaque valeur affichée.

Il se peut que l'on ne désire pas passer à la ligne après un appel à `print`, ou que l'on ne veuille pas avoir une espace comme séparateur entre les valeurs.

Il est possible de modifier ces deux comportements de la fonction `print`, en explicitant ce que l'interpréteur doit utiliser comme chaîne de caractères de séparation (argument nommé `sep`) et de fin d'affichage (argument nommé `end`).

Par exemple :

```
date_1 = 1515
date_2 = 1789
print("Bataille de Marignan", date_1, sep = ' : ', end = '/')
print("Révolution française", date_2, sep = ' : ', end = '\n-----\n')
```

affiche les deux lignes

```
Bataille de Marignan : 1515/Révolution française : 1789
-----
```

3.5.6 Exercice UpyLaB 3.11 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire un programme qui lit sur input une valeur naturelle n et qui affiche à l'écran un carré de n caractères X (majuscule) de côté.

Exemple 1

Avec la donnée lue suivante :

```
6
```

le résultat à imprimer vaudra :

```
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
XX
XX
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple).
- Il n'est pas demandé de tester si la valeur n est bien positive ou nulle, vous pouvez supposer que ce sera toujours le cas pour les valeurs transmises par UpyLaB.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Si votre programme est rejeté par UpyLaB alors qu'il semble produire le résultat attendu, vérifiez bien que vous n'ajoutez pas d'espaces superflus, en début ou en fin de ligne, et que votre code n'affiche pas une ligne vide supplémentaire.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.11*.

3.5.7 Exercice UpyLaB 3.12 (Parcours Bleu et Rouge)

Énoncé

Cet exercice propose une variante de l'exercice précédent sur le carré de X .

Écrire un programme qui lit sur `input` une valeur naturelle n et qui affiche à l'écran un triangle supérieur droit formé de X (voir exemples plus bas).

Exemple 1

Avec la donnée lue suivante :

```
6
```

le résultat à imprimer vaudra :

```
XXXXXX
 XXXXX
  XXXX
   XXX
    XX
     X
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
XX
 X
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple).
- Il n'est pas demandé de tester si la valeur n est bien positive ou nulle, vous pouvez supposer que ce sera toujours le cas pour les valeurs transmises par UpyLaB.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- La concaténation (qui consiste à coller deux textes ensemble) et l'opérateur `*` sur les chaînes de caractères pourront s'avérer utiles ici.
- Si votre programme est rejeté par UpyLaB alors qu'il semble produire le résultat attendu, vérifiez bien que vous n'ajoutez pas d'espaces superflus, en début ou en fin de ligne, et que votre code n'affiche pas une ligne vide supplémentaire.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.12*.

3.5.8 Exercice UpyLaB 3.13 (Parcours Bleu et Rouge)

Énoncé

Écrire un programme qui additionne des valeurs naturelles lues sur `input` et affiche le résultat.

La première donnée lue ne fait pas partie des valeurs à sommer. Elle détermine si la liste contient un nombre déterminé à l'avance de valeurs à lire ou non :

- si cette valeur est un nombre positif ou nul, elle donne le nombre de valeurs à lire et à sommer ;
- si elle est négative, cela signifie qu'elle est suivie d'une liste de données à lire qui sera terminée par le caractère "F" signifiant que la liste est terminée.

Exemple 1

Avec les données lues suivantes :

```
4
1
3
5
7
```

qui indiquent qu'il y a 4 données à sommer : $1 + 3 + 5 + 7$,

le résultat à imprimer vaudra donc

```
16
```

Exemple 2

Avec les données lues suivantes :

```
-1
1
3
5
7
21
F
```

qui indiquent qu'il faut sommer : $1 + 3 + 5 + 7 + 21$,

le résultat à imprimer vaudra donc

```
37
```

Exemple 3

Avec la donnée :

```
0
```

qui indique qu'il faut sommer 0 nombre,

le résultat à imprimer vaudra donc

```
0
```

Consignes

- Ne mettez pas d'argument dans l'input :
- `data = input()` et pas
- `data = input("Donnée suivante : ")` par exemple ;
- Le résultat doit juste faire l'objet d'un `print(res)` sans texte supplémentaire (pas de `print("résultat =", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Dans la cas où la liste est terminée par le caractère "F", envisagez de lire l'input, puis de tester si la valeur lue est différente du caractère "F", avant de chercher à la convertir en int et à l'ajouter à ce qui a déjà été sommé ;
- Utilisez un `for` dans le cas où le nombre de valeurs à sommer est connu, un `while` dans le cas contraire.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.13*.

3.5.9 Exercice UpyLaB 3.14 (Parcours Rouge)

Énoncé

Dans cet exercice, nous revenons sur le petit jeu de devinette.

Écrire un programme qui génère de manière (pseudo) aléatoire un entier (nombre secret) compris entre 0 et 100. Ensuite, le joueur doit deviner ce nombre en utilisant le moins d'essais possible.

À chaque tour, le joueur est invité à proposer un nombre et le programme doit donner une réponse parmi les suivantes :

- « Trop grand » : si le nombre secret est plus petit que la proposition et qu'on n'est pas au maximum d'essais
- « Trop petit » : si le nombre secret est plus grand que la proposition et qu'on n'est pas au maximum d'essais
- « Gagné en n essais ! » : si le nombre secret est trouvé
- « Perdu ! Le secret était *nombre* » : si le joueur a utilisé **six essais** sans trouver le nombre secret.

Exemple 1

Une partie gagnante (après la génération du nombre à deviner) :

NB : Les nombres sont les valeurs saisies par l'utilisateur, et les textes sont imprimés par le programme.

```
50
Trop grand
8
Trop petit
20
Trop petit
27
Gagné en 4 essais !
```

Exemple 2

Une partie gagnante (après la génération du nombre à deviner) :

```
50
Trop grand
24
Trop petit
37
Trop petit
43
Trop grand
40
Trop petit
41
Perdu ! Le secret était 42
```

Consignes

- Attention, au dernier essai, le programme ne doit afficher ni « Trop petit » ni « Trop grand », mais le verdict comme illustré plus haut.
- Pour qu'Upylab puisse tester que votre solution est correcte, il faut que vous respectiez strictement la séquence décrite dans l'énoncé. Si par exemple, vous n'affichez pas « Trop petit » ou « Trop grand », le nombre suivant ne sera pas fourni par le système et votre solution sera considérée comme incorrecte.
- En pratique, pour la génération du nombre secret, vous devez débiter votre code comme suit :

```
import random
NB_ESSAIS_MAX = 6
secret = random.randint(0, 100)
```

et ne pas faire d'autre appel à `randint` ou à une autre fonction du module `random`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester notre code, UpyLaB va exécuter le programme en ajoutant l'instruction `random.seed(argument)`, où `argument` est une certaine valeur. Cette instruction permet de générer les mêmes suites de nombres aléatoires lors des appels aux fonctions du module `random` à chaque exécution du programme.
Si, avec PyCham, vous souhaitez reproduire le comportement exact d'UpyLaB lors des tests, il faudra donc rajouter cette instruction juste après l'import du module `random`, en remplaçant `argument` par la valeur indiquée dans le test d'UpyLaB. Vous serez ainsi sûr que votre code générera le même nombre secret qu'UpyLaB, ce qui peut faciliter le débogage en cas de test invalidé par exemple.
Notez bien que l'argument de la fonction `seed` n'est pas le nombre à deviner, mais un paramètre qui permet de le générer. Attention, cette instruction ne doit pas figurer dans le code que vous soumettez à UpyLaB.
- Si vous rencontrez l'erreur `EOF Error`, vérifiez que votre programme se termine bien dès que l'utilisateur découvre le nombre secret et ce, même en moins de 6 essais, ou bien dès que le nombre maximal d'essais est atteint.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.14*.

3.5.10 Exercice UpyLaB 3.15 (Parcours Rouge)

Énoncé

Comme mes écureuils s'ennuyaient, je leur ai fabriqué une roue avec des barreaux pour qu'ils puissent faire de l'exercice.

La roue fait 100 barreaux. Pour m'y retrouver, je les numérote de 0 à 99.

Très vite je me suis rendu compte que chacun utilisait la roue en sautant toujours le même nombre de barreaux (Up saute 7 barreaux à chaque fois, Py en saute 9, LaB en saute 13, ...).

Je mets une noisette sur un des barreaux de la roue. Aidez moi à savoir si un de mes écureuils va l'attraper sachant que je vais mettre l'écureuil qui fait le test, par exemple Up, sur le barreau 0 et que je connais son comportement (Up saute toujours 7 barreaux à la fois) et que je sais le numéro de barreau, différent de 0, où se trouve la noisette (Up n'aura donc pas la noisette au départ).

Écrire un programme qui teste si pour une configuration donnée, l'écureuil va ou non atteindre un moment la noisette. Il reçoit deux valeurs entières en entrée, une valeur `saut` et une valeur `position_cible` toutes deux entre 1 et 99.

Le programme va calculer une valeur `position_courante`, initialement la valeur 0, et vérifier si en calculant de façon répétitive la valeur `position_courante`, celle-ci aboutira un moment à la valeur `position_cible`.

Notez que pour calculer la valeur suivante de `position_courante` (initialement mise à 0), il faut incrémenter la valeur actuelle de `position_courante` de la valeur `saut` et ensuite, si le résultat est plus grand ou égal à 100, calculer la position en faisant un modulo 100 de la valeur obtenue (ce qui donne à chaque fois une valeur `position_courante` entre 0 et 99). (Notez que l'on peut systématiquement faire le modulo 100 du résultat sans tester si `position_courante` est ou non supérieur à 100 pour obtenir sa bonne valeur).

Notez également, pour ne pas épuiser mon écureuil sans fin, que s'il atteint à nouveau la barreau 0, j'arrête l'expérience sachant qu'il prendra toujours les mêmes barreaux sans jamais atteindre `position_cible` (la noisette).

À la fin votre programme dira si oui ou non la noisette a été atteinte ou non.

En pratique, après avoir lu les deux valeurs `saut` et `position_cible`, votre programme affichera chaque valeur de `position_courante` sur une ligne différente à partir de la seconde valeur (pas la `position_courante` initiale qui vaut toujours 0). La dernière `position_courante` affichée sera soit 0 soit la dernière valeur de `position_courante` avant qu'elle n'ait la valeur de `position_cible`, si l'écureuil trouve la noisette. Votre programme terminera en affichant, sur une nouvelle ligne, le message donnant le résultat :

- "Cible atteinte" si l'écureuil a trouvé la noisette,
- "Pas trouvé" si l'écureuil est revenu en position 0 sans trouver la noisette.

Vous pouvez supposer que les valeurs lues sont bien des entiers qui respectent les consignes.

Exemple 1

Avec les données lues suivantes :

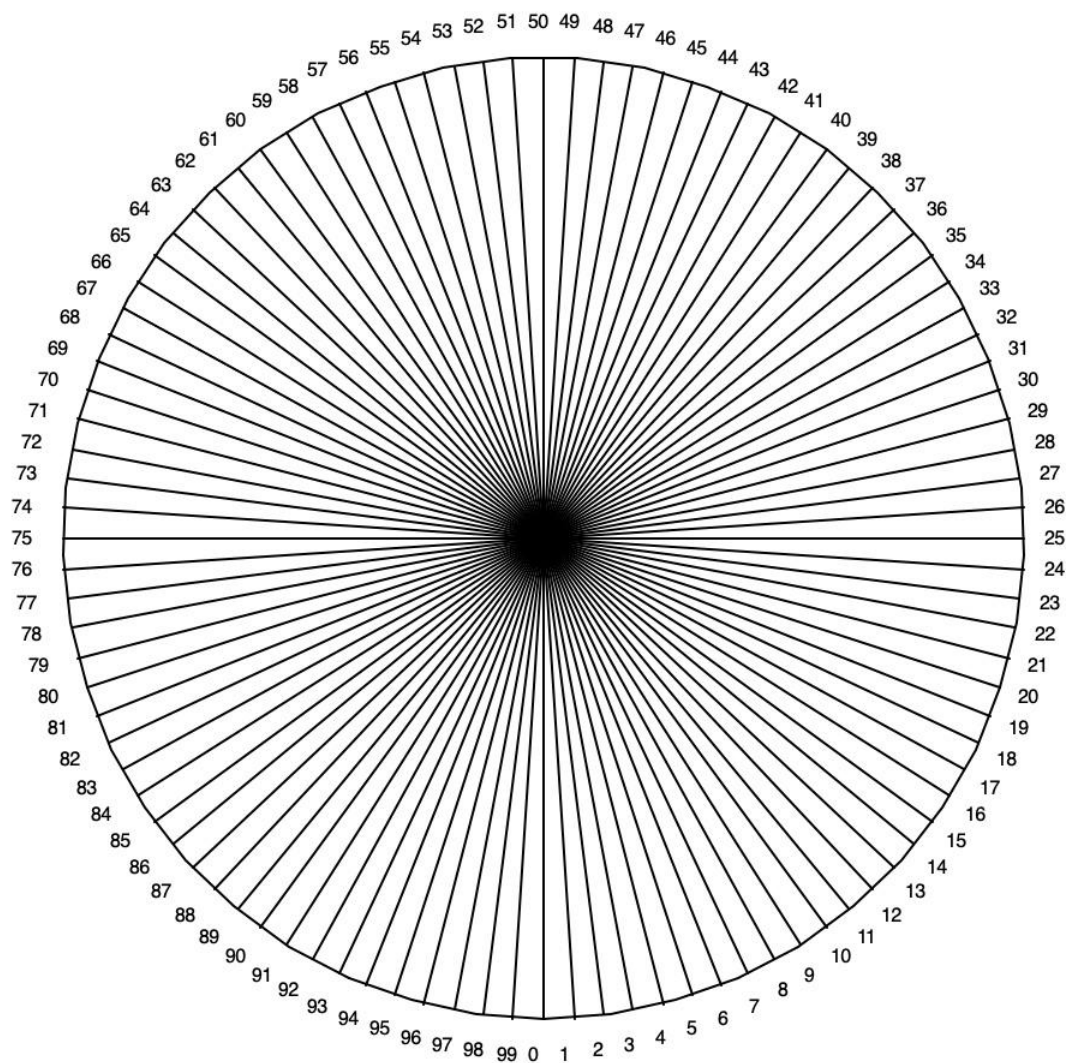


Fig. 3.7 – Roue de mes écureuils

```
9
7
```

le résultat à imprimer vaudra :

```
9
18
27
36
45
54
63
72
81
90
99
8
17
26
35
44
53
62
71
80
89
98
Cible atteinte
```

Exemple 2

Avec les données lues suivantes :

```
8
7
```

le résultat à imprimer vaudra :

```
8
16
24
32
40
48
56
64
72
80
88
96
4
12
20
28
36
44
52
60
```

(suite sur la page suivante)

(suite de la page précédente)

```
68
76
84
92
0
Pas trouvé
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), et à afficher précisément le texte demandé; par exemple : `print("Cible atteinte")` et non `print("La cible a été atteinte")` par exemple.

CONSEILS

- Il sera utile de définir trois variables, une qui stockera la valeur `saut` (inchangée tout au long du programme puisqu'elle est utilisée comme incrément), une qui stockera la valeur `position_cible` où se trouve la noisette, et enfin une qui stockera la valeur `position_courante`. **Nous vous conseillons** au début d'assigner à cette variable `position_courante` la valeur `saut` soit la position le l'écureuil après un saut (la position initiale de l'écureuil valant 0).
- Nous rappelons que l'opérateur modulo (`%` en Python) donne le reste de la division euclidienne. Ainsi, `42 % 5` est égal à 2 car $42 = 5 * 8 + 2$.
- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la [FAQ sur UpyLaB 3.15](#).

3.5.11 range(debut, fin, pas)

RANGE

Nous avons déjà utilisé `range` lors de la présentation de l'instruction `for` à la section précédente. Expliquons plus complètement les différentes formes d'utilisation de cette instruction.

`range` donne une séquence de valeurs entières. On l'utilise en lui donnant de 1 à 3 arguments entiers.

range avec un argument

Nous avons vu l'utilisation avec un seul argument, comme avec :

```
for i in range(5):
    print(i)
```

l'instruction `range` affiche

```
0
1
2
3
4
```

soit une séquence avec les valeurs entières depuis 0 compris jusque 5 non compris.

range avec deux arguments

Si nous utilisons dans un code `range` avec deux arguments, le premier argument spécifie la première valeur de la séquence (si la séquence n'est pas vide), et le deuxième argument la borne finale (non incluse). Par exemple :

```
somme = 0
for i in range(1,11):
    somme = somme + i
print(somme)
```

affiche

```
1
3
6
10
15
21
28
36
45
55
```

soit les différentes valeurs de `somme` sommant en 10 étapes les valeurs entières de 1 à 10.

range avec trois arguments

Si nous utilisons dans un code `range` avec trois arguments, les deux premiers arguments spécifient la première valeur de la séquence et la borne finale non comprise. Le troisième argument donne le *pas* ou *incrément* pour calculer la valeur suivante à partir de la valeur précédente. Par exemple :

```
for i in range(0, 51, 5):
    print(i, end = ' ')
print()
```

affiche les différentes valeurs de `i` soit :

```
0 5 10 15 20 25 30 35 40 45 50
```

depuis la valeur 0 jusqu'à la valeur 51 non comprise par pas de 5 (à chaque étape, pour avoir la valeur suivante, on incrémente la précédente de 5).

Dans cet exemple les multiples de 5, entre 0 et 50 compris, sont affichés sur la même ligne et séparés d'une espace grâce à l'argument nommé `end = ' '`; et après la dernière valeur affichée, le `print()` placé après l'instruction `for`, permet de passer à la ligne suivante pour tout affichage ultérieur.

Notons que :

- la valeur du *pas* peut être négative (par exemple : `range(10, -1, -1)`)
- `range` peut donner une séquence vide (par exemple : `range(0)`, `range(1, 1)`, `range(0, 10, -1)`)

3.5.12 Exercice UpyLaB 3.16 (Parcours Rouge)

Énoncé

Ecrivez un code qui lit un nombre entier strictement positif n et affiche sur n lignes une table de multiplication de taille $n \times n$, avec, pour i entre 1 et n , les n premières valeurs multiples de i strictement positives sur la i ème ligne. Ainsi, les n premiers multiples de 1 strictement positifs (0 non compris) sont affichés sur la première ligne, les n premiers multiples de 2 sur la deuxième, et caetera.

Exemple 1

Avec la valeur lue suivante :

le résultat à afficher sera :

```
1  2  3
2  4  6
3  6  9
```

Exemple 2

Avec la valeur lue suivante :

le résultat à afficher sera :

```
1  2  3  4  5  6  7  8  9 10
2  4  6  8 10 12 14 16 18 20
3  6  9 12 15 18 21 24 27 30
4  8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Exemple 3

Avec la valeur lue suivante :

le résultat à afficher sera :

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé.
- Pour cet exercice, UpyLaB ne tiendra pas compte du nombre d'espaces séparant les nombres sur chaque ligne, ni de la présence d'espace en fin de ligne.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- L'utilisation de `range(...)` peut être utile.

3.5.13 Exercice UpyLaB 3.17 (Parcours Rouge)

Énoncé

On peut calculer approximativement le sinus d'un nombre x en effectuant la sommation des premiers termes de la série (une série est une somme infinie) :

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

où x est exprimé en radians et $3!$ désigne la factorielle de 3.

Écrire un programme qui lit une valeur flottante x en entrée et imprime une approximation de $\sin(x)$.

Cette approximation sera obtenue en additionnant successivement les différents termes de la série jusqu'à ce que la valeur du terme devienne inférieure (en valeur absolue) à une constante ϵ que l'on fixera à 10^{-6} .

Exemple 1

Avec les données lues suivantes :

0.8

le résultat à imprimer vaudra :

0.7173557231746032

Remarque : Compte-tenu du manque de précision concernant les calculs sur les float, vous pourrez obtenir un résultat sensiblement différent. Ce n'est pas un problème, car UpyLaB acceptera toute réponse suffisamment proche de celle attendue, avec une tolérance de l'ordre de $1.0e-5$.

Exemple 2

Avec les données lues suivantes :

-0.5

le résultat à imprimer vaudra :

-0.479425533234127

Consignes

- Nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et `non int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et `non print("résultat : ", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Dans cet exercice, notre programme doit calculer une approximation du sinus d'un nombre mais il ne faut pas utiliser la fonction `sin` du module `math`.
- Pour éviter de calculer explicitement la valeur des factorielles, on pourra chercher à exprimer chacun des termes en fonction du précédent.
- Notez bien que les exposants de x ne sont que les nombres impairs, et que le signe alterne entre $+$ et $-$.

3.5.14 Exercice UpyLaB 3.18 (Parcours Rouge)

Énoncé

Écrire un code qui lit un nombre entier strictement positif n et imprime une pyramide de chiffres de hauteur n (sur n lignes complètes, c'est-à-dire toutes terminées par une fin de ligne).

- La première ligne imprime un “1” (au milieu de la pyramide).
- La ligne i commence par le chiffre $i \% 10$ et tant que l'on n'est pas au milieu, le chiffre suivant a la valeur suivante $((i+1) \% 10)$.
- Après le milieu de la ligne, les chiffres vont en décroissant modulo 10 (symétriquement au début de la ligne).

Notons qu'à la dernière ligne, aucune espace n'est imprimée avant d'écrire les chiffres 0123 . . .

Exemple 1

Avec la donnée lue suivante :

```
1
```

le résultat à imprimer vaudra :

```
1
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
1
232
```

Exemple 3

Avec la donnée lue suivante :

```
10
```

le résultat à imprimer vaudra :

```
      1
     232
    34543
   4567654
  567898765
 67890109876
7890123210987
890123454321098
90123456765432109
0123456789876543210
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à `input(int(input()))` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans PyCharm avec des valeurs différentes en entrée y compris supérieure à 10.

3.6 L'instruction pass et quiz de fin de module

3.6.1 L'instruction pass

L'INSTRUCTION PASS QUI NE FAIT RIEN

Avant de terminer ce module, nous voudrions vous présenter l'instruction `pass`. L'instruction `pass` est une instruction simple, même très simple.

Sa syntaxe unique est

```
pass
```

et elle a comme effet, quand l'interpréteur l'exécute, de ne rien faire.

Dans ce cas, à quoi bon introduire une telle instruction ?

En fait, le programmeur utilise généralement l'instruction `pass` quand il est en phase de développement de son programme, et qu'il n'a pas encore écrit certaines parties de son code.

Par exemple ayant une certaine valeur pour `x`

```
if x < 0 :  
    pass # TODO compléter le code (cas où x < 0)  
else :  
    print('traitement du cas où x est positif')
```

l'instruction `pass`, éventuellement avec un commentaire du type `A_FAIRE` (ou `TO DO` en anglais), permet d'avoir un code syntaxiquement correct tout en soulignant qu'une partie du code reste à écrire.

3.6.2 Quiz de fin de module

QUIZ POUR RÉCAPITULER CE QUE L'ON A VU DANS CE MODULE

Dans ce module nous avons appris comment fonctionne la plupart des instructions de contrôle de flux Python.

Pour bien assimiler la matière, analysons quelques bouts de code pour déterminer ce qu'ils font.

QUIZ

Note : Voir le quiz de la section 3.6.2

3.7 Bilan du module

3.7.1 Qu'avons-nous vu dans ce module ?

BILAN EN BREF

Note : Voir la vidéo de la section 3.7.1 : Bilan du module

BILAN DU MODULE

Nous voici à la fin de ce module.

Nous y avons vu principalement comment contrôler la séquence des instructions en fonction de conditions. De façon plus précise, nous avons vu :

- que les conditions sont exprimées sous forme de valeurs et d'expressions booléennes qui après évaluation peuvent avoir une valeur vraie (`True` en anglais) ou fausse (`False` en anglais)
- que les opérateurs relationnels comparent les valeurs : `<`, `<=`, `>`, `>=`, `==` et `!=`
- que l'on peut connecter des expressions booléennes avec les opérateurs booléens `and`, `or` et `not` ; nous avons vu leur syntaxe et leur sémantique ainsi que les lois de De Morgan permettant de simplifier certaines expressions booléennes
- l'instruction conditionnelle `if` avec possibilité de parties `elif` et d'une partie `else`
- l'instruction `pass`
- l'instruction répétitive `while` appelée également boucle `while`
- la fonction prédéfinie `range` qui permet de générer une séquence de nombres entiers
- l'instruction répétitive `for` qui traite chaque élément d'une séquence

Nous avons également illustré tous ces concepts avec de nombreux exemples et la réalisation d'exercices supervisés ou réalisés de façon autonomes avec UpyLaB.

En route pour le module suivant qui va vous montrer comment définir vos propres fonctions Python.

Les fonctions : créons les outils que nous voulons

4.1 Au menu : comment fonctionne une fonction ?

4.1.1 Présentation du menu de ce module

MENU DE CE MODULE : EN BREF

Python permet de découper les programmes grâce à des fonctions. Dans ce module nous allons vous expliquer comment définir et utiliser une fonction dans un code Python et comment écrire des programmes modulaires, ce qui signifie ici des programmes dont le code est découpé en parties avec des fonctions. Comme dans le module précédent, nous passerons de sections théoriques d'introduction à des sections de mise en pratique des concepts, d'abord supervisées et ensuite que vous réaliserez de façon autonome.

MENU DE CE MODULE

Note : Voir la vidéo de la section 4.1.1 : Les fonctions

4.2 Les fonctions prédéfinies et définies

4.2.1 Utilisation et définition de fonctions

FONCTIONS PRÉDÉFINIES

Note : Voir la vidéo de la section 4.2.1 : Les fonctions prédéfinies et leur utilisation

CONTENU DE LA VIDÉO

La vidéo précédente présente plusieurs fonctions Python prédéfinies, utilisables directement par l'interpréteur ou après leur importation par exemple des modules `math`, `random` ou `time`.

Code sur la console réalisé dans la vidéo

```
>>> x = 666
>>> y = 7
>>> z = 3.14159
>>> abs(x)
666
>>> abs(-10)
10
>>> dir(x)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
↳ '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__
↳ ge__', '__getattr__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__
↳ init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__
↳ __', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__
↳ rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__
↳ __', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__
↳ rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__
↳ __', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes
↳ __', 'imag', 'numerator', 'real', 'to_bytes']
>>> divmod(x, y)
(95, 1)
>>> float(x)
666.0
>>> input()
>? 1111
'1111'
>>> int(z)
3
>>> max(x, y, 1515)
1515
>>> min(x, 1551, y+3, 27)
10
>>> round(z, 3)
3.142
>>> type(z)
<class 'float'>
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.

>>> divmod(x, y)
(95, 1)
>>> 666 // 7
95
>>> 666 % 7
1
>>> print("x vaut:", x)
x vaut: 666
>>> print()

>>> from math import pi, cos, log
>>> cos(pi/4)
```

(suite sur la page suivante)

(suite de la page précédente)

```

0.7071067811865476
>>> log(1024,2)
10.0
>>> help(log)
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.

>>> log(3)
1.0986122886681098
>>> log(100,10)
2.0
>>> import random
>>> random.randint(1, 100)
77
>>> import time
>>> time.sleep(5)

>>>

```

La vidéo en bref

Nous illustrons l'utilisation des fonctions prédéfinies `abs`, `dir`, `divmod`, `float`, `input`, `int`, `max`, `min`, `round`, `type`, `help` et `print`.

Nous montrons ce que réalisent certaines fonctions ou valent certaines constantes des modules `math` (`log`, `cos`, `pi`), `random` (`randint`) et `time` (`sleep`).

4.2.2 Définition de nouvelles fonctions

DÉFINITION DE NOUVELLES FONCTIONS

Note : Voir la vidéo de la section 4.2.2 : Définition d'une nouvelle fonction

CONTENU DE LA VIDÉO

La vidéo précédente illustre, à partir de l'exemple du pgcd, comment on peut créer ses propres fonctions en Python.

Script réalisé dans la vidéo

```

def pgcd(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
        x, y = y, x % y
    return x

print('le pgcd de 112 et de 30 vaut : ', pgcd(112, 30))
a = int(input(" a = "))

```

(suite sur la page suivante)

(suite de la page précédente)

```
if pgcd(a, 6) == 2:
    print("le pgcd est égal à 2")
else:
    print("le pgcd est différent de 2")
```

La vidéo en bref

Nous avons présenté la syntaxe d'une définition de fonction sur l'exemple de la fonction `pgcd` : avec la ligne d'entête suivie d'un docstring expliquant ce que reçoit la fonction comme paramètres et ce qu'elle fait et du corps de la fonction terminée par l'instruction `return`.

Après avoir défini la fonction `pgcd`, la suite du code peut en faire usage.

La séquence d'exécution lors de la définition et de l'appel à une fonction est détaillée dans l'unité suivante.

4.2.3 Exécution de fonctions

SÉQUENCE D'EXÉCUTION D'UNE FONCTION

Note : Voir la vidéo de la section 4.2.3 : Exécution d'une fonction

CONTENU DE LA VIDÉO

La vidéo précédente présente la séquence d'exécution lors de l'appel à une telle fonction Python.

Script réalisé dans la vidéo

```
def pgcd(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
        x, y = y, x % y
    return x

print('le pgcd de 112 et de 30 vaut : ', pgcd(112, 30))
a = int(input(" a = "))
if pgcd(a, 6) == 2:
    print("le pgcd est égal à 2")
else:
    print("le pgcd est différent de 2")
```

La vidéo en bref

Lorsqu'une fonction est appelée par une autre fonction ou dans le code « principal » (que nous appelons le code « appelant »), la séquence du traitement est la suivante :

- 1) Les variables locales à la fonction appelée, correspondantes aux paramètres formels, sont créées (et mises dans le nouvel « espace de nom » associé à cette instance de la fonction).
- 2) Les paramètres sont transmis (passés) entre le code appelant et la fonction appelée. Ceci est décrit plus en détails à la sous-section suivante.
- 3) L'exécution du code appelant est suspendue pour laisser la place à l'exécution de la fonction appelée.

- 4) La fonction appelée s'exécute éventuellement en créant de nouvelles variables et des objets.
- 5) Lorsque la fonction appelée termine son exécution soit quand l'instruction `return` est exécutée, soit après l'exécution de la dernière instruction de la fonction, les variables locales à la fonction sont "détruites" : plus précisément l'espace de nom associé à l'instance de la fonction est détruit.
- 6) Le code appelant reprend son exécution. Si la fonction précédemment appelée était une fonction retournant une valeur avec l'instruction `return`, la valeur de retour a été conservée pour permettre l'évaluation de l'expression utilisant le résultat de cette fonction.

4.2.4 Précisions sur les fonctions

PRÉCISIONS SUR LES FONCTIONS ET LE MÉCANISME DE PASSAGE DE PARAMÈTRES

Notons qu'un *espace de nom* est l'endroit où des variables locales sont stockées, sachant qu'en Python une variable est un nom pour un objet.

Il doit y avoir une correspondance entre les paramètres formels et les paramètres effectifs selon la position. Cela signifie que le i-ème paramètre effectif en ordre d'apparition dans la liste doit correspondre au i-ème paramètre formel.

Le passage de paramètres suit le même fonctionnement qu'une assignation : un paramètre effectif soit désigne un objet existant soit est une expression dont la valeur donne un objet ; le paramètre formel correspondant est une variable locale à la fonction qui lors de son exécution, sera un nom local de cet objet.

Donc avec `def pgcd(x, y) :` l'appel `pgcd(26, a)` aura comme effet que dans l'instance de la fonction `pgcd`, `x` désigne l'objet entier valant 26 et `y`, la valeur de la variable nommée par ailleurs `a`.

Le mot instance de fonction signifie exemplaire ; en effet une fonction peut être appelée plusieurs fois, et peut même s'appeler elle-même « récursivement » ce qui entraîne plusieurs instances simultanées lors de l'exécution. La récursivité ne sera pas abordée dans ce cours.

COMMENT ET POURQUOI DÉFINIR ET UTILISER DES FONCTIONS ?

Pour définir une nouvelle fonction, il faut écrire son **entête** qui commence par le mot-clé « `def` » suivi du nom de la fonction et des paramètres appelés dans le jargon standard **paramètres formels** entourés de parenthèses, le tout terminé par le caractère deux-points (« : »).

Le **corps** de la fonction est identifié par l'ensemble des lignes de code qui suivent indentées par rapport à l'entête.

```
def nom (paramètres formels):
    corps de la fonction
```

L'utilisation de la fonction se fait quand une instruction donne le nom de cette fonction avec les **paramètres effectifs**, également appelés **paramètres réels** ou **arguments**, correspondants.

```
nom(arguments)
```

Chaque fonction peut être vue comme un outil résolvant un certain problème. Les autres fonctions peuvent alors utiliser cet outil sans se soucier de sa structure, juste pour le résultat qu'il donne. Pour pouvoir l'utiliser, il leur suffit de savoir comment l'employer, c'est-à-dire de connaître le nom de la fonction, les valeurs qu'elle nécessite et la façon dont elle renvoie un ou plusieurs résultats.

Cette technique a, en particulier, deux avantages :

- chaque fonction peut (généralement) être testée indépendamment du reste du programme ;
- si une partie contient une séquence d'instructions qui doit être réalisée à différents endroits du programme, il est possible de n'écrire cette séquence qu'une seule fois.

En gros, une fonction peut être vue comme une opération dont la valeur est définie par le programmeur en fonction des paramètres reçus.

4.2.5 Mini quiz

MINI QUIZ SUR LES FONCTIONS

Avant d'examiner de plus près les possibilités et le fonctionnement des fonctions Python, ce quiz va vous aider à vérifier que vous avez bien assimilé ce que nous avons vu jusqu'ici.

Note : Voir le quiz de la section 4.2.5 du cours en ligne

4.3 Faisons fonctionner les fonctions

4.3.1 Autres exemples de fonctions

AUTRES EXEMPLES DE FONCTIONS AVEC PARAMÈTRES

Donnons d'autres exemples de fonctions, qui varient en particulier selon les paramètres reçus et le ou les résultat(s) renvoyé(s) :

Fonction booléenne : La fonction `est_pair` est une fonction booléenne, c'est-à-dire dont le résultat est une valeur booléenne, qui renvoie *vrai* (`True`) si la valeur reçue en paramètre est paire.

```
def est_pair(x):
    """Renvoie vrai si et seulement si x est pair."""
    return x % 2 == 0

a = int(input("Entrez un nombre entier impair : "))
if est_pair(a):
    print("Il n'est pas impair !")
```

Fonction qui renvoie None : La fonction `print_line` affiche un résultat sur l'écran (l'output) mais ne retourne pas de valeur.

```
def print_line(x, n):
    """Imprime x*n."""
    print(x*n)
```

Par défaut, la fonction renvoie la valeur `None`. Il s'agit d'une valeur d'un type à part (`NoneType`) qui symbolise l'absence de valeur. Cela signifie que l'instruction `return` sans rien derrière équivaut à `return None`. Par ailleurs, un `return` implicite existe à la fin de toute fonction pour revenir au code appelant (avec la valeur de retour `None`).

Fonction qui renvoie un tuple : La fonction `min_et_max` renvoie deux valeurs :

```
def min_et_max(x, y):
    """Renvoie min(x, y) suivi de max(x, y)."""
    if x > y:
        x, y = y, x
    return x, y
```

Plus précisément, toute fonction Python renvoie une seule valeur à la fin de son exécution. Ici `min_et_max` renvoie en réalité un objet de type tuple qui contient deux valeurs. Nous verrons dans le prochain module, qui traite des séquences, cette notion de tuple plus en détails.

UN EXEMPLE DE FONCTION SANS PARAMÈTRE

Fonction sans paramètre : Notons que nous pouvons définir une fonction sans paramètre. Dans ce cas, dans l'entête de cette dernière, il faut quand même mettre des parenthèses sans rien dedans. Par exemple, le code suivant utilise `turtle`, et en particulier

la fonction `circle`, pour dessiner un symbole Yin et yang. Pour ce faire, nous utilisons la fonction prédéfinie `turtle.circle` avec un ou deux paramètres :

`turtle.circle(rayon [, angle])` :

- trace un cercle ou un arc de cercle de `|rayon|` (valeur absolue de rayon);
- le cercle est complet si le paramètre `angle` n'existe pas;
- le cercle ne fait qu'un arc de `angle` degrés si `angle` est donné.

Par exemple `turtle.circle(100, 180)` trace un demi cercle (soit 180 degrés) de rayon de 100 points :

- si `rayon` est positif, le cercle est tracé dans le sens positif, c'est-à-dire anti-horlogique, sinon (`rayon` est négatif), il est tracé dans le sens horlogique;
- si `angle` est positif, la tortue avance, sinon elle recule.

Le code ci-dessous donne une solution à la fonction `yin_yang()` :

```
""" Exemple de code sans paramètre :
yin et yang dessiné avec turtle
"""
import turtle

def yin_yang():
    """ dessine un logo yin-yang de rayon 200 """
    turtle.down() # met la plume en mode tracé (si ce n'était déjà le cas)
    turtle.width(2) # grosseur du tracé de 2 points
    # dessine le yin externe
    turtle.color("black", "black") # le tracé et le remplissage seront en noir
    turtle.begin_fill() # la ou les formes suivantes seront remplies
    turtle.circle(-100, 180) # demi cercle intérieur tournant vers la droite
    turtle.circle(-200, -180) # demi cercle extérieur, en marche arrière
    turtle.circle(-100, -180) # demi cercle intérieur qui complète le yin
    turtle.end_fill() # remplissage
    # dessine le yang interne
    turtle.color("white") # couleur blanche
    turtle.up() # on ne trace pas ce qui suit
    # déplace la tortue au bon endroit
    turtle.right(90)
    turtle.forward(80)
    turtle.left(90)
    # tracé du disque yang (blanc) interne au yin
    turtle.down()
    turtle.begin_fill()
    turtle.circle(-20)
    turtle.end_fill()
    # se replace au centre
    turtle.up()
    turtle.left(90)
    turtle.forward(80)
    turtle.right(90)
    # dessine le yang externe
    turtle.down()
    turtle.color("black", "white") # contour noir, remplissage blanc
    turtle.begin_fill()
    turtle.circle(-100, 180)
    turtle.circle(-200, -180)
    turtle.circle(-100, -180)
    turtle.end_fill()
    # tracé du disque yin (noir) interne au yang

    turtle.color("black")
    # déplace la tortue au bon endroit
    turtle.up()
    turtle.right(90)
```

(suite sur la page suivante)

(suite de la page précédente)

```
turtle.forward(80)
turtle.left(90)
turtle.down()
# trace le disque
turtle.begin_fill()
turtle.circle(-20)
turtle.end_fill()
# se replace au centre
turtle.up()
turtle.left(90)
turtle.forward(80)
turtle.right(90)
turtle.down()
turtle.hideturtle()
return

#code principal
yin_yang() #réalise le logo
```

L'exécution du code ci-dessus donne le résultat montré par la petite vidéo donné en section 4.3.1 du cours en ligne.

DESSIN D'UN YIN-YANG AVEC TURTLE

Note : Voir la vidéo de la section 4.3.1 du cours en ligne : Dessin d'un yin-yang avec turtle

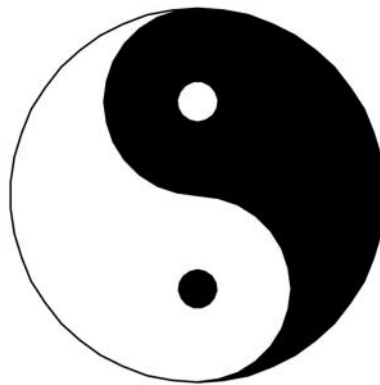


Fig. 4.1 – Dessin de yin-yang est réalisé avec turtle

COMPLÉMENT D'INFORMATIONS

Notons dans le code précédent l'utilisation de `turtle.color(couleur1, couleur2)` qui demande que les tracés aient les contours en couleur1 ("black" par exemple) et les surfaces remplies (grâce à `turtle.begin_fill()` et `turtle.end_fill()`) en couleur2 ("white" par exemple pour faire le yang).

Nous discuterons de la qualité de ce code dans la section suivante qui parle de règles de bonnes pratiques pour coder ; nous y verrons pourquoi un programmeur expérimenté n'aurait pas écrit la fonction `yin_yang` telle quelle.

4.3.2 Quelques aspects plus techniques

Avant de mettre en pratique les fonctions, les points suivants, plus techniques, doivent être mis en évidence car ils sont importants pour ne pas écrire de codes erronés.

POLYMORPHISME ET CONTRÔLE DE TYPE

Même si en pratique ce n'est généralement pas le cas, en théorie une fonction Python peut recevoir différents types de paramètres et renvoyer des résultats de types différents : c'est ce que l'on appelle la surcharge des paramètres.

Par exemple avec la définition :

```
def sum(a,b):
    """renvoie la somme ou concaténation des deux paramètres"""
    return a+b
```

les 3 appels successifs :

```
print(sum(3,5))
print(sum('cha', 'peau'))
print(sum(3, 'peau'))
```

renvoient respectivement :

```
8
chapeau
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    File "<input>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

La fonction `sum` peut donc faire des sommes de valeurs, des concaténations de texte (qui consistent à coller deux textes ensemble) ou même produire des erreurs, suivant le type des arguments utilisés. Dans le jargon informatique, on parle de polymorphisme, ce qui signifie que quelque chose, ici une fonction, de même forme, peut en fait représenter plusieurs choses ou comme ici, avoir des effets différents.

En Python, il est donc généralement fort conseillé, si l'on n'est pas sûr du code appelant, que la fonction vérifie elle-même que les arguments donnés aient un type adéquat pour son exécution.

La section suivante explique une bonne façon de faire cela.

ASSERT

L'instruction `assert` vérifie si une certaine condition est rencontrée. Dans l'exemple précédent, si les seuls types permis pour les paramètres de notre fonction `sum` sont des `int` ou des `float`, une instruction à ajouter en début du code de la fonction juste après l'entête et le docstring de la fonction est :

```
assert (type(a) is int or type(a) is float) and (type(b) is int or type(b) is float)
```

qui, lors de son exécution, teste la condition ; si elle est vraie, l'interpréteur passe à l'instruction suivante, sinon il provoque une erreur qui indique que l'assertion est fausse.

VARIABLES GLOBALES ET LOCALES

Dans la section précédente, nous avons vu que lors de son exécution, la fonction appelée utilise des variables qui lui sont locales. Nous avons vu comment s'effectue le passage de paramètres entre les arguments effectifs et les paramètres formels. Par ailleurs, le code principal utilise des variables globales que l'interpréteur continue à manipuler après l'exécution de la fonction. La présente

unité revient en détails sur les mécanismes qui gèrent les variables : en effet, nous devons bien les comprendre, sous peine d'écrire des codes qui ne font pas du tout ce que nous désirons. Allons-y !

Assignment : Nous avons vu en section 2.7 qu'après les instructions :

```
a = b = 3
b = 2 * b
```

la variable `b` vaut 6 tandis que la variable `a` vaut toujours 3.

Plus précisément, après la première instruction, `a` et `b` sont deux noms pour l'objet entier valant 3, comme le montre le premier diagramme d'état ci-dessous. Ensuite, l'évaluation de `2 * b` crée un nouvel objet entier valant 6 et `b` référence 6 comme le montre le second diagramme d'état après la seconde instruction.

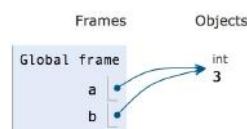


Fig. 4.2 – Diagramme d'état après `a = b = 3`

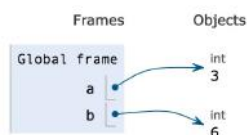


Fig. 4.3 – Diagramme d'état après `b = 2 * b`

Pour vous en convaincre complètement, vous pouvez exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Voir la première animation Python Tutor en section 4.3.2 du cours en ligne

Exécution étape par étape du code avec Python Tutor

Le passage de paramètres utilise le même mécanisme comme le montre l'exemple suivant :

```
def fun(x):
    x = x**2
    return x

a = 3
fun(a)
print(a)
```

On peut voir que la variable `a` n'a pas été modifiée par la fonction comme le montre le diagramme d'état au moment du `return`.

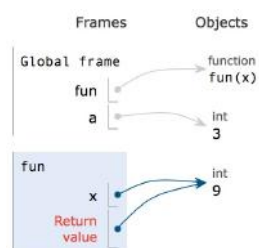


Diagramme d'état au moment du `return x`

À nouveau pour vous en convaincre complètement, vous pouvez exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Exécution étape par étape du code avec Python Tutor : voir cours en ligne section 4.3.2

Nom des variables globales et locales

Même si il est mieux d'éviter cela pour des raisons de clarté du code global, nous pouvons vérifier que le fait qu'une variable locale porte le même nom qu'une variable globale ne change rien. Le code qui suit est une modification de l'exemple précédent mais manipule une variable globale `x` ainsi qu'une autre variable locale à la fonction `fun`, qui porte le même nom `x`.

```
def fun(x):
    x = x**2
    return x

x = 3
fun(x)
print(x)
```

Nous pouvons exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Voir la troisième animation Python Tutor en section 4.3.2 du cours en ligne

Les animations Python Tutor présentées précédemment montrent clairement qu'une variable locale, comme le paramètre formel `x` de la fonction `fun`, n'est pas connue en dehors de l'exécution de l'instance de fonction où elle est créée. De plus à la fin de l'exécution de cette instance de fonction, les variables locales à cette instance de fonction sont supprimées. Ce qui nous amène à la question suivante :

Pourquoi les fonctions Python utilisent-elles des variables locales ?

Quand le programmeur écrit une fonction `fun`, l'échange de valeurs et de résultats entre la fonction `fun` et la fonction (ou le programme global) est réalisé via les mécanismes du passage des paramètres et par l'instruction `return`. Pour que les codes de la fonction et du programme soient « propres », il faut éviter toute autre inférence : c'est ce qui est réalisé par le mécanisme des variables locales qui permet d'isoler proprement le code de la fonction `fun` (utilisant uniquement des variables locales qui lui sont propres) du code qui utilise cette fonction `fun` et d'autres variables.

Attention, jusqu'à présent, aucun type d'objets présenté dans ce cours n'est modifiable. En effet, par exemple

```
x = 1
x = 2
```

crée un premier objet de type `int` valant 1 nommé `x` via l'affectation et ensuite un second objet de même type valant 2 et le nom `x` est donné à ce second objet via la seconde affectation ; ce qui rend le premier objet inaccessible puisqu'il n'a plus de nom. Un nom de variable peut donc désigner des objets différents tout au long de l'exécution, mais dans cet exemple les objets eux-mêmes de type `int` sont non modifiables.

Nous verrons dans le prochain module que certains objets (par exemple les listes) sont modifiables. Dans ce cas, si c'est requis pour la fonction `fun`, le mécanisme de passage de paramètres permettra à la fonction de modifier un paramètre, c'est-à-dire un objet reçu du code appelant la fonction `fun`. Notez cependant que si aucune modification de paramètre n'est demandée, une règle de bonne pratique est de veiller à ne pas modifier ceux-ci dans le code de la fonction. Les exemples plus haut de fonction `fun` qui modifient le paramètre formel `x` sont donc à éviter.

Certains mécanismes permettent de déroger au mécanisme de variables locales et globales en permettant, par exemple, à une fonction de manipuler des variables globales. Comme ce n'est généralement pas utile ni une bonne pratique pour obtenir un code « propre », nous vous recommandons de ne pas utiliser ces mécanismes et de n'utiliser que des variables locales. Dans ce cours, seules des constantes globales seront utilisées, comme nous le verrons à la section suivante.

UN PEU DE VOCABULAIRE : PORTÉE ET VISIBILITÉ

Les variables qui sont créées et ensuite supprimées font appel à deux concepts : portée et index : *visibilité*. Expliquons ces termes utilisés fréquemment par les programmeurs.

Dans le jargon informatique, l'ensemble des endroits où dans un programme, une variable existe est appelé sa portée (scope en anglais). L'ensemble des endroits où une variable est visible est appelé sa visibilité.

Dans l'exemple précédent, une variable globale, comme `x`, existe depuis sa création jusqu'à la fin de l'exécution du programme, mais n'est pas visible dans la fonction `fun` : en l'occurrence la variable `x` locale cache la variable `x` globale.

TRACEBACK

Nous avons déjà rencontré le mot Traceback lorsque notre code s'arrêtait en produisant une erreur. Le Traceback est une information que l'interpréteur donne au programmeur pour qu'il trouve son erreur de code. Une traduction possible de ce verbe est retracer. C'est donc une trace de l'exécution au moment de l'erreur. Expliquons la notion de Traceback sur un exemple. L'exécution du code ci-dessous (où nous avons rajouté les numéros de ligne pour que ce soit plus clair), comme script PyCharm sauvé dans mon répertoire Desktop par exemple,

```
1 def ma_fun() :  
2     une_autre_fun()  
3  
4 def une_autre_fun() :  
5     encore_une_fun()  
6  
7 def encore_une_fun() :  
8     # une_erreur est utilisé ligne 9 sans avoir été définie !  
9     une_erreur = une_erreur + 1  
10  
11 ma_fun()
```

génère le message d'erreur suivant quand l'interpréteur essaye d'exécuter l'instruction `une_erreur = une_erreur + 1` alors que la variable `une_erreur` n'est pas encore définie (aucune assignation n'a été effectuée avant).

```
Traceback (most recent call last):  
  File "/Users/tmassart/Desktop/test_4_1_2018.py", line 11, in <module>  
    ma_fun()  
  File "/Users/tmassart/Desktop/test_4_1_2018.py", line 2, in ma_fun  
    une_autre_fun()  
  File "/Users/tmassart/Desktop/test_4_1_2018.py", line 5, in une_autre_fun  
    encore_une_fun()  
  File "/Users/tmassart/Desktop/test_4_1_2018.py", line 9, in encore_une_fun  
    une_erreur = une_erreur + 1  
UnboundLocalError: local variable 'une_erreur' referenced before assignment
```

Le message d'erreur explique le type de l'erreur (en dernière ligne) tandis que les lignes précédentes donnent un résumé du « chemin » pris lors de l'exécution pour arriver à l'instruction qui a causé cette erreur. On peut ainsi retracer la séquence des appels en cours quand l'erreur s'est produite.

4.3.3 Première mise en pratique des fonctions

LE N-ÈME NOMBRE DE FIBONACCI

Après cette longue suite d'explications, il est grand temps de mettre en pratique les nouveaux concepts que nous avons vus dans ce module.

Reprenons le principe du calcul des nombres de Fibonacci présenté dans le module 3, section 3.5.1.

Rappelons que les premiers nombres de Fibonacci sont :

0 1 1 2 3 5 8 13 21 34...

où après les deux premiers qui sont donnés, la valeur des nombres suivants est donnée en sommant les deux précédents.

Plus mathématiquement, nous pouvons dénoter les nombres de Fibonacci par :

$$F_0 = 0 \quad (4.1)$$

$$F_1 = 1 \quad (4.2)$$

$$F_{i+1} = F_{i-1} + F_i, \text{ pour } i > 0 \quad (4.3)$$

Supposons que notre programme utilise régulièrement des nombres de Fibonacci. Une solution simple est d'écrire une fonction qui calcule et renvoie F_n pour l'indice n désiré.

Commençons par choisir un nom à notre fonction et déterminer le ou les paramètre(s) formel(s).

Les règles de bonnes pratiques pour encoder des programmes Python, comme discutées dans la section suivante, demandent que les noms des fonctions, comme celui des variables, débutent par une lettre alphabétique en minuscule et soient formés de lettres alphabétiques minuscules, de chiffres ou du caractère souligné '_'. Dans le jargon, cette convention s'appelle « snake case ».

Appelons notre fonction `fibonacci` et le paramètre formel `n` qui correspond à l'indice du nombre de Fibonacci à calculer.

```
def fibonacci(n):
```

Ensuite, écrivons le docstring en indiquant ce que fait `fibonacci` et le contenu initial de `n` au moment de chaque appel :

```
"""calcule le n-ième nombre de Fibonacci, avec : n de type int et
fibonacci(0) valant 0
fibonacci(1) valant 1 et
fibonacci(n+1) valant fibonacci(n-1) + fibonacci(n)
si n < 0 : fibonacci(n) retourne None"""
```

Après, le corps de la fonction sera écrit et terminé par un `return` du résultat.

Il nous reste à :

- déterminer les hypothèses. Par exemple supposons, sans devoir le tester, que `n` est de type `int` ;
- identifier les différents cas (`n < 0`, `n == 0`, `n == 1`, `n > 1`) ;
- déterminer ce que la fonction renvoie dans chacun des cas.

Enfin, **après** cette définition de la fonction `fibonacci(n)`, nous pouvons écrire le code qui utilise cette fonction.

À VOUS DE JOUER !

Ouvrez un nouveau script dans PyCharm et complétez le code pour avoir une définition complète de la fonction `fibonacci(n)`.

Un conseil : pour le cas général, une instruction répétitive pourrait être utile.

Ensuite, en dessous de la définition, ajoutez un code qui teste la fonction, par exemple en imprimant le résultat pour les valeurs entre 0 et 100 non compris.

Votre code aura donc la structure suivante :

```
def fibonacci(n):
    """calcule le n-ième nombre de Fibonacci, avec : n de type int et
    fibonacci(0) valant 0
    fibonacci(1) valant 1 et
    fibonacci(n+1) valant fibonacci(n-1) + fibonacci(n)
    si n < 0 : fibonacci(n) retourne None"""
    ...
    code correspondant au corps de la fonction fibonacci
    ...
```

(suite sur la page suivante)

(suite de la page précédente)

```
return res

for i in range(100):
    print(fibo(i))
```

4.3.4 Mise en pratique des fonctions : exercices UpyLaB 4.1 et suivants

MISE EN PRATIQUE AVEC UPYLAB

Faites les exercices UpyLaB 4.1 à 4.4 qui suivent. Notez que l'exercice UpyLaB 4.4 est très proche du programme et de la fonction `fibonacci` demandés précédemment.

EXERCICE UPYLAB 4.1 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire une fonction `deux_egaux(a, b, c)` qui reçoit trois nombres en paramètre et qui renvoie **la valeur booléenne** `True` si au moins deux de ces nombres ont la même valeur, et **la valeur booléenne** `False` sinon.

Ensuite, **écrire un programme** qui lit trois données de type `int`, `x`, `y` et `z`, et affiche le résultat de l'exécution de `deux_egaux(x, y, z)`.

Exemple 1

Avec les données lues suivantes :

```
1
2
3
```

le résultat à imprimer vaudra donc

```
False
```

Exemple 2

Avec les données lues suivantes :

```
42
6
42
```

le résultat à imprimer vaudra donc

```
True
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire une fonction, puis un programme appelant cette fonction sur des valeurs lues en entrée. Notez qu'UpyLaB testera ces deux points, en exécutant le programme entier mais aussi en appelant directement la fonction avec les arguments de son choix.

- Il n'est pas demandé que la fonction `deux_egaux` teste le type des paramètres reçus.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Par contre quand UpyLaB teste spécifiquement le code d'une fonction (ici la fonction `deux_egaux`), le type (et on verra plus tard aussi la structure) du résultat est également validé. Veillez donc bien à renvoyer un résultat de type requis.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Veillez à ce que la fonction retourne bien les valeurs booléennes `True` ou `False`, et non les chaînes de caractères `"True"` ou `"False"`.
- La fonction doit retourner `True` si au moins deux nombres passés en paramètre sont égaux, que doit alors retourner l'appel `deux_egaux(2, 2, 2)` ?
- Si rien ne marche : consultez la [FAQ sur UpyLaB 4.1](#).

4.3.5 Exercice UpyLaB 4.2 (Parcours Vert, Bleu et Rouge)

Énoncé

Attention : cet exercice est composé de l'exercice UpyLaB 4.2.a suivi en dessous de l'exercice UpyLaB 4.2.b.

Le Petit Prince vient de débarquer sur la planète U357, et il apprend qu'il peut y voir de belles aurores boréales !

La planète U357 a deux soleils : les étoiles E1515 et E666. C'est pour cela que les tempêtes magnétiques sont permanentes, ce qui est excellent pour avoir des aurores boréales.

Par contre, il y fait souvent jour sauf bien évidemment quand les deux soleils sont couchés en même temps.

Heureusement pour nous, une journée U357 s'écoule sur 24 heures comme sur notre Terre, et pour simplifier, nous ne prendrons pas en compte les minutes (on ne donne que les heures avec des valeurs entières entre 0 et 23).

Nous vous demandons d'aider le Petit Prince à déterminer les périodes de jour et de nuit.

Pour cela, vous allez dans un premier temps écrire une fonction `soleil_leve` qui, pour un soleil particulier, reçoit trois valeurs

- l'heure actuelle (entre 0 et 23)
- l'heure de lever du soleil (entre 0 et 23)
- l'heure du coucher du soleil (entre 0 et 23)

et qui renvoie une valeur booléenne vraie si le soleil est levé sur la planète à l'heure donnée en premier argument et fausse, s'il est couché.

On supposera que chacun des soleils ne se lève et ne se couche au plus qu'une seule fois par jour. Il est toutefois possible que le lever ait lieu après l'heure du coucher, ce qui signifie dans ce cas que le soleil est levé au début de la journée, puis qu'il se couche, puis qu'il se lève à nouveau plus tard dans la journée. Enfin, si l'heure du lever est la même que l'heure du coucher :

- soit toutes deux valent 12, cela signifie que le soleil ne se lève pas de la journée,
- soit toutes les deux valent 0, cela signifie que le soleil ne se couche pas de la journée.

Exemple 1

L'appel de la fonction suivant :

```
soleil_leve(6, 18, 10)
```

doit retourner

True

Exemple 2

L'appel de la fonction suivant :

```
soleil_leve(15, 8, 12)
```

doit retourner

False

Exemple 3

L'appel de la fonction suivant :

```
soleil_leve(12, 12, 10)
```

doit retourner

False

Exemple 4

L'appel de la fonction suivant :

```
soleil_leve(0, 0, 22)
```

doit retourner

True

Consignes

- Il n'est pas demandé que la fonction `soleil_leve` teste le type du paramètre reçu.
- La fonction `soleil_leve` ne doit rien afficher.
- Quand UpyLaB teste spécifiquement le code d'une fonction (ici la fonction `soleil_leve`), le type du résultat est également validé. Veuillez donc bien à renvoyer un résultat de type requis. En particulier, les objets `True` et `"True"` ne sont pas du même type.

Il vous faut maintenant **écrire un programme** qui lit en entrée :

. l'heure de lever du soleil E1515 . l'heure du coucher du soleil E1515 . l'heure de lever du soleil E666 . l'heure du coucher du soleil E666

et qui utilise la fonction `soleil_leve` pour afficher ligne par ligne chacune des heures de la journée, depuis 0 jusqu'à 23, suivies d'une espace et d'une astérisque s'il fait nuit à cette heure.

Attention, il ne fera nuit que si E1515 et E666 sont tous deux couchés.

Exemple 1

Avec les données lues suivantes :

```
6
18
10
21
```

le résultat à imprimer vaudra donc

```
0 *
1 *
2 *
3 *
4 *
5 *
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 *
22 *
23 *
```

Exemple 2

Avec les données lues suivantes :

```
15
8
6
17
```

le résultat à imprimer vaudra donc

```
0
1
2
3
4
5
6
7
8
9
10
```

(suite sur la page suivante)

(suite de la page précédente)

```
11
12
13
14
15
16
17
18
19
20
21
22
23
```

Consignes

- N’oubliez pas d’insérer votre fonction `soleil_leve`.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu’il affiche, donc veillez à n’imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l’intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

4.3.6 Exercice UpyLaB 4.3 (Parcours Bleu et Rouge)

CALCULS DE NOMBRES PREMIERS : FONCTION PREMIER(N)

L’exercice 4.3 d’UpyLaB vous propose une fonction assez classique à rédiger, qui teste si un nombre `n`, reçu en paramètre, est premier.

Par définition, un nombre entier naturel est premier s’il est divisible (entièrement) uniquement par deux nombres différents, par 1 et par lui-même. 1 n’est pas premier. Le plus petit nombre premier est 2. Tous les nombres premiers suivants sont impairs puisque tous les nombres pairs sont divisibles par 2.

Il faut rédiger une fonction booléenne (dont le résultat est `True` ou `False`), qui reçoit un entier positif et renvoie `True` si et seulement si `n` est un nombre premier.

L’exercice n’a pas d’applications pratiques dans ce cours, mais comme le problème est assez simple, c’est l’occasion d’essayer d’être le plus succinct possible. L’extrait de code suivant peut servir de base. Il vous reste à compléter le corps de la fonction `premier`.

```
def premier(n):
    """ renvoie vrai si n est un nombre premier """

    ...
    return res

# code principal
print("liste des nombres premiers jusqu'à 100")
for i in range(100):
    if premier(i):
        print(i)
```

MODULO

Dans la fonction `premier`, il faut vérifier que `n` n’est divisible par aucun des nombres entiers à partir de 2 et strictement inférieurs à lui-même. Pour cela, notons que l’opérateur modulo peut être utile : le test `n % i == 0` est vrai si `i` divise `n` entièrement.

NB : Il est toutefois intéressant de constater qu'il n'est en fait nécessaire de vérifier cette condition que pour les nombres entiers qui sont inférieurs ou égaux à la racine carrée du nombre n . Si vous souhaitez rendre votre code plus efficace en réduisant le nombre d'instructions exécutées, la fonction `sqrt` du module `math` s'avèrera utile ici.

ÉNONCÉ EXERCICE UPYLAB 4.3

Rédigez dans un script PyCharm une fonction `premier` qui réalise le traitement expliqué précédemment, et le code qui l'appelle. Ensuite, pour valider votre solution, modifiez votre code pour répondre à l'exercice 4.4 d'UpyLaB dont l'énoncé est donné ci-dessous.

EXERCICE UPYLAB 4.3 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire une fonction booléenne `premier(n)` qui reçoit un nombre entier positif n et qui renvoie `True` si n est un nombre premier, et `False` sinon.

Ensuite, écrire un programme qui lit une valeur entière x et affiche, grâce à des appels à la fonction `premier`, tous les nombres premiers **strictement inférieurs** à x , chacun sur sa propre ligne.

Exemple 1

Avec la donnée lue suivante :

7

le résultat à imprimer vaudra donc

2
3
5

Exemple 2

Avec la donnée lue suivante :

9

le résultat à imprimer vaudra donc

2
3
5
7

Consignes

- Dans cet exercice, il vous est demandé d'écrire une fonction, puis un programme appelant cette fonction. Notez qu'UpyLaB testera ces deux points, en exécutant le programme entier mais aussi en appelant directement la fonction avec les arguments de son choix.
- Il n'est pas demandé que la fonction `premier` teste le type du paramètre reçu, ni si sa valeur est bien positive ou nulle.

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer, ou pour les fonctions ne renvoyer, que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour le programme principal, ne pas hésiter à s'inspirer de ce qui est proposé dans le texte accompagnant cet exercice.
- Attention, le programme ne doit pas afficher la valeur de x , même si celui-ci est premier.
- Pour tester si un nombre est premier, l'idée est de chercher, parmi les nombres qui lui sont inférieurs (0 et 1 exclus), un éventuel diviseur. Une instruction itérative, une instruction conditionnelle `if` et l'opérateur `%` (modulo) s'avèreront donc utiles.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.3*.

4.3.7 Exercice UpyLaB 4.4 (Parcours Bleu et Rouge)

Énoncé

Écrire une fonction `fibonacci(n)` qui reçoit un nombre entier n et qui renvoie la valeur du nombre de Fibonacci F_n .

On rappelle que :

- F_0 vaut 0 ;
- F_1 vaut 1 ;
- F_{i+1} vaut $F_i + F_{i-1}$ pour $i > 0$;
- F_i vaut *None* pour $i < 0$.

Ensuite, écrire un programme qui lit une valeur entière strictement positive x et affiche le résultat de `fibonacci(i)` pour i allant de 0 compris à x non compris, avec chaque valeur sur sa propre ligne.

Exemple 1

Avec la donnée lue suivante :

5

le résultat à imprimer vaudra donc

0
1
1
2
3

Exemple 2

Avec la donnée lue suivante :

1

le résultat à imprimer vaudra donc

0

Consignes

- Dans cet exercice, il vous est demandé d'écrire une fonction, puis un programme appelant cette fonction. Notez qu'UpyLaB testera ces deux points, en exécutant le programme entier mais aussi en appelant directement la fonction avec les arguments de son choix.
- Il n'est pas demandé que la fonction `fibonacci` teste le type du paramètre reçu.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Par ailleurs, la fonction `fibonacci` est demandée; UpyLaB va valider qu'en appelant la fonction indépendamment du reste de votre code, les résultats envoyés par votre fonction sont corrects et ont le bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Ne pas hésiter à s'inspirer du code déjà écrit lors de l'activité guidée dans un précédent onglet de cette section.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.4*.

4.4 Quelques règles de bonnes pratiques

4.4.1 Quiz “Testez vos connaissances !”

Note : Voir le quiz de la section 4.4.1 du cours en ligne

4.4.2 Code ou programme « propre »

QU'EST-CE QU'UN CODE « PROPRE »

Jusqu'à présent nous avons appris de nouveaux éléments montrant ce qui était possible avec le langage Python. Parfois, il faut aussi parler de ce qui est possible mais doit être évité. La parallèle peut être faite avec l'apprentissage de la vie en société. C'est bien de savoir que l'on peut bouger, communiquer, et satisfaire ses besoins divers, encore faut-il connaître le code de la bonne conduite en société pour ne pas avoir des soucis avec les autres ou se faire ostraciser !

De même, quand un programmeur écrit un code pour résoudre un problème, il faut d'abord que ce code soit **correct**, c'est-à-dire résolve le problème pour toutes les configurations possibles (données, contextes, ...). Ensuite il vaut mieux que le code soit **efficace** en prenant le moins de temps possible et en occupant le moins de place mémoire lors de son exécution.

Pour bien programmer, un autre critère essentiel est que le programme ou le code soit « **propre** » : **un code est propre** s'il a un certain nombre de qualités de lisibilité pour un programmeur, de structuration, s'il est bien commenté, ... Python édicte des règles à suivre au mieux lors de l'écriture du code pour que ce dernier soit propre. Libre à chaque programmeur de suivre ces règles ou pas. Mais dans ce dernier cas, ne vous étonnez pas que personne ne communique avec vous vu les difficultés pour les autres de comprendre votre code !

Plus globalement, les recommandations et améliorations pour le langage Python sont répertoriées dans des documents appelés PEPs (Python Enhancement Proposals) sur le site officiel <https://www.python.org>. Chaque PEP porte un numéro. Le document intitulé « PEP 8 – Style Guide for Python Code » contient un certain nombre de ces « **règles de bonnes pratiques** ». Ne pas suivre ces

règles peut aboutir à un programme correct et efficace, mais probablement moins propre. Ces règles parlent de constantes globales, de convention de codage, de taille des fonctions, de conventions typographiques (endroits où l'on met des espaces ou des lignes blanches), ... Nous en parlons dans ce module et le suivant, mais donnons directement un exemple de code qui suit ces règles.

Constantes globales

Nous avons vu qu'un programme complet pouvait contenir des définitions de fonctions, du code, des commentaires, mais également des importations de fonctions prédéfinies ou de modules complets. Généralement, un programme utilise également des valeurs constantes, plus simplement appelées constantes dans le jargon informatique. Ces constantes peuvent être un « paramètre » du programme, par exemple le nombre maximum d'éléments à traiter ou une valeur approchée de pi (3.14159...). Reprenons le petit jeu proposé dans le module précédent en section 3.3, pour améliorer le code proposé. Si notre programme doit deviner un nombre entier entre 0 et 5 que nous avons choisi en utilisant la fonction `randint` du module `random`, il peut être intéressant de donner un nom aux valeurs minimales et maximales de l'intervalle, et donc de définir des « constantes globales » :

```
VALEUR_MIN = 0      # borne inférieure de l'intervalle
VALEUR_MAX = 5      # borne supérieure de l'intervalle
```

Ces constantes seront bien sûr utilisées ensuite tout au long du programme. Un des gros avantages de cette pratique est que si plus tard, le programmeur veut changer cet intervalle de valeurs, il lui suffit de corriger ces deux lignes en début du code.

Structure d'un programme

Les règles de bonnes pratiques nous demandent qu'un programme complet soit formé dans l'ordre :

- du docstring initial du programme contenant, le nom de l'auteur, la date d'écriture, une brève explication de ce qu'il fait, des entrées lues et résultats affichés ;
- des éventuels imports de modules ou de parties de modules ;
- des définitions des constantes globales ;
- des définitions des fonctions globales ;
- et enfin du code du traitement principal.

La découpe du code en fonctions est également importante pour sa clarté. Donnons-en un simple exemple.

Code amélioré du petit jeu de devinette : Continuons avec l'exemple du petit jeu proposé dans le module précédent (section 3.3). Le code suivant vous montre un exemple complet où nous supposons que :

- la valeur aléatoire est choisie dans l'intervalle [VALEUR_MIN, VALEUR_MAX] ;
- l'utilisateur propose bien une valeur entière, mais peut se tromper et proposer une valeur en dehors de l'intervalle.

Par ailleurs, par souci de clarté, nous avons découpé le code grâce à des fonctions :

- une fonction `tirage` qui s'occupe du tirage aléatoire ;
- une fonction `entree_utilisateur` qui demande à l'utilisateur son choix et redemande si ce dernier ne propose pas un nombre dans l'intervalle ;
- une fonction `affichage_resultat` qui affiche à l'écran si l'utilisateur a ou non trouvé le nombre tiré aléatoirement.

L'exemple qui suit respecte bien l'ordre dans lequel les différents éléments de code doivent être placés : import, définitions de constantes globales, de fonctions et finalement code principal.

Nous noterons qu'il respecte également les autres règles prescrites dont nous parlerons plus loin.

```
"""
Petit jeu de devinette (version 2)
Auteur: Thierry Massart
Date : 10 octobre 2018
Petit jeu de devinette d'un nombre entier tiré aléatoirement
par le programme dans un intervalle donné
Entrée : le nombre proposé par l'utilisateur
Résultat : affiche si le nombre proposé est celui tiré
           aléatoirement
"""

# importation des modules

import random # module le tirage des nombres aléatoires
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Définition des constantes globales

VALEUR_MIN = 0 # borne inférieure de l'intervalle
VALEUR_MAX = 5 # borne supérieure de l'intervalle

# Définition des fonctions

def entree_utilisateur(borne_min, borne_max):
    """
    Lecture du nombre entier choisi par l'utilisateur
    dans l'intervalle [borne_min, borne_max]
    Entrées : bornes de l'intervalle
    Résultat : choix de l'utilisateur
    """
    message = "Votre choix de valeur entre {0} et {1} : "
    ok = False # vrai quand le choix donné est valable
    while not ok: # répétition tant que le choix n'est pas bon
        choix = int(input(message.format(borne_min, borne_max)))
        ok = (borne_min <= choix and choix <= borne_max)
        if not ok: # entrée hors de l'intervalle
            print("Hors de l'intervalle ! Donnez une valeur valide")
    return choix

def tirage(borne_min, borne_max):
    """
    Tirage aléatoire d'un entier dans [borne_min, borne_max]
    """
    return random.randint(borne_min, borne_max)

def affichage_resultat(secret, choix_utilisateur):
    """
    Affiche le résultat
    """
    if secret == choix_utilisateur:
        print("gagné !")
    else:
        print("perdu ! La valeur était", secret)

# Code principal

mon_secret = tirage(VALEUR_MIN, VALEUR_MAX)
choix_util = entree_utilisateur(VALEUR_MIN, VALEUR_MAX)
affichage_resultat(mon_secret, choix_util)
```

Note sur format()

Le code utilise la fonction (appelée méthode dans le jargon) `format` pour imprimer du texte lors d'un input (ici pour demander le choix de l'utilisateur). Cette méthode peut servir, lors d'un print, à formater les impressions, par exemple si l'on veut imprimer un float avec un certain nombre de chiffres après le point décimal.

Notons que l'instruction Python `"Votre choix de valeur entre {0} et {1} : ".format(10, 100)` donne le texte où les « paramètres » {0} et {1} dans le texte ont été remplacés par les valeurs 10 et 100 donnés comme arguments dans la méthode `format`.

Pour voir son effet plus précis, essayez par exemple, dans une console PyCharm ou de votre environnement Python 3 :

```
print("pi vaut plus ou moins {0:7}".format(3.14))
print("pi vaut plus ou moins {0:7.4}".format(3.14159265359))
print("pi vaut plus ou moins {0:07.4}".format(3.14159265359))
```

Si vous désirez avoir une connaissance approfondie de son fonctionnement, reportez-vous au [manuel python](#) qui malheureusement est souvent un peu ardu à lire.

Note sur les f-strings

À partir de Python 3.6, la notion de f-string a été introduite pour remplacer avantageusement la méthode `format()`. Malheureusement, la version actuelle d'UpyLaB est 3.5.3 et ne supporte donc pas les f-strings. Les f-strings ne seront pas vus dans ce cours.

4.4.3 Quelques éléments supplémentaires

ENCORE QUELQUES CONVENTIONS DE CODAGE

Rajoutons quelques règles de bonnes pratiques pour avoir un code « propre ». Nous avons déjà vu que parmi celles-ci :

- une convention habituelle en Python pour choisir un nom à une variable ou à une fonction est que ce nom soit formé de lettres alphabétiques en minuscules, de chiffres ou du caractère souligné '_' et qu'elle évoque le contenu qu'elle représente. Par exemple, `resultat_final` est un nom de variable correct pour recevoir un résultat ;
- si on définit une constante, c'est-à-dire que l'on donne un nom à une valeur que l'on ne modifie pas tout au long de l'exécution du programme, la convention pour donner un nom à cette constante est d'utiliser des majuscules et le caractère souligné, par exemple `DIM_MAX = 100` définit la constante `DIM_MAX` (on suppose que le programmeur ne réassigne pas une autre valeur à `DIM_MAX` ailleurs dans le programme).

En matière de bonnes pratiques, voici encore quelques éléments indispensables :

Indentation : La règle simple est d'ajouter 4 caractères pour chaque nouvelle indentation plutôt que d'utiliser la tabulation. Un bon éditeur de script, par exemple celui fourni dans PyCharm, vous aidera grandement pour cela en faisant la plupart du travail automatiquement.

Lignes de continuation : Les lignes doivent normalement avoir moins de 79 caractères (72 pour les Docstrings). Les blocs de texte plus longs s'écrivent sur plusieurs lignes. Il se peut que le caractère de continuation anti-slash (en anglais backslash) '`\`' en fin de ligne soit nécessaire, même si la plupart du temps, ce caractère de continuation peut être évité quand l'interpréteur sait que l'instruction ou l'expression sur la ligne a une suite.

Par exemple, ayant dans une console PyCharm :

```
>>> ma_variable_dont_le_nom_est_trop_long = 0
```

Si nous voulons incrémenter cette variable, la ligne suivante donnera une erreur de syntaxe :

```
>>> ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long +  
... 1  
File "<ipython-input-3-be9f0e911bbc>", line 1  
ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long +  
                                         ^  
SyntaxError: invalid syntax
```

Une solution simple est d'ajouter des parenthèses à l'expression à droite :

```
>>> ma_variable_dont_le_nom_est_trop_long = (ma_variable_dont_le_nom_est_trop_long +  
... 1)
```

ou le caractère de continuation :

```
>>> ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long + \  
... 1
```

ou plus simplement utiliser l'opérateur d'incrémementation :

```
>>> ma_variable_dont_le_nom_est_trop_long += 1
```

Lignes blanches : Les règles de bonnes pratiques conseillent de mettre deux lignes blanches avant et entre chacune des définitions de fonction globales et une ligne blanche pour mettre en évidence une nouvelle partie dans une fonction.

DÉFINITION DE FONCTIONS IMBRIQUÉES

Une dernière remarque dans cette activité : en Python, une fonction `fun_2` peut être définie à l'intérieur d'une autre fonction `fun_1` à condition que `fun_2` soit locale à `fun_1`, c'est-à-dire utilisée uniquement lors du traitement de `fun_1`. Par exemple, dans la fonction `pgcd(x, y)`, on pourrait définir la fonction `suisant` (même si dans ce cas, cela ne simplifie pas fort le code) comme suit :

```
def pgcd(x, y):
    """ calcule le plus grand commun diviseur de deux entier positifs """

    def suisant(x, y):
        """ calcule le couple de valeurs suivantes dans pgcd """
        return (y, x%y)

    # code de pgcd
    while y > 0:
        x, y = suisant (x, y)
    return x
```

La fonction `suisant` n'est utilisée que par `pgcd` : en dehors de l'exécution de `pgcd`, `suisant` n'existe pas. Notons que certains langages de programmation connus, tels que les langages C, C++, Java, ne supportent pas les définitions de fonctions imbriquées.

4.4.4 Petit manuel des règles de bonnes pratiques

PETIT MANUEL À AVOIR À PORTÉE DE CLAVIER

Vous n'avez évidemment pas vu toutes les notions utiles pour écrire un programme Python. Et probablement vous n'avez pas intégré toutes les règles que nous venons de présenter. Ce n'est pas grave ! Pour vous aider, voici un petit manuel que vous pouvez télécharger en cliquant sur le lien [petit manuel des bonnes pratiques de programmation \(Python\)](https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@BonnesPratiquesPython.pdf) <<https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@BonnesPratiquesPython.pdf>>, qui donne les règles de bonnes pratiques que nous voudrions que vous respectiez pour coder proprement vos programmes Python.

Pourquoi ne pas imprimer cette page pour toujours l'avoir sous la main quand vous écrirez du code ?

Ce manuel d'une page est découpé en cinq parties :

- traduction d'un problème en programme ;
- programmation ;
- nommage de variables, fonctions, etc. ;
- documentation du code ;
- structure globale d'un programme.

Chaque partie de ce manuel est essentielle et doit être suivie pour veiller à obtenir le code le plus lisible possible.

- La traduction d'un problème en programme passe par une phase d'analyse et est suivie de codage où il faut choisir les fonctions que nous désirons définir et utiliser.
- La partie programmation parle de style de programmation et de quelques erreurs classiques à éviter.
- Nous avons déjà parlé de conventions de nommage des variables, des fonctions, etc.
- La documentation du code signifie que, dans vos programmes, vous devez mettre des commentaires (docstrings ou commentaires en fin de ligne, ...) pour que, si quelqu'un ou même vous, plus tard, relisez le code, il sera facile de comprendre ce qu'il fait. La documentation est donc inutile pour l'ordinateur, et en particulier pour l'interpréteur, mais est bien utile pour les pauvres humains que nous sommes qui essayons de comprendre ce que fait un code !
- Enfin, la dernière partie rappelle et illustre la structure globale d'un programme.

Ce petit manuel est également donné ci-dessous : à vous de lire et d'apprendre l'ensemble des consignes !

Une dernière bonne nouvelle : l'éditeur de script PyCharm intègre la plupart de ces règles et vous indique là où il considère que vous ne les respectez pas.

LE MANUEL DE BONNES PRATIQUES

Le petit manuel des bonnes pratiques Python est accessible en <https://www.fun-mooc.fr/asset-v1:ulb+44013+session03+type@asset+block@BonnesPratiquesPython.pdf>

4.4.5 La règle du return unique et les instructions dont on ne veut pas prononcer le nom

RÈGLES POUR NE PAS ENTRER PAR LA PORTE ET SORTIR PAR LA FENÊTRE

Pour ceux, comme vous probablement, qui débutent l'apprentissage de la programmation, nous avons l'habitude de rajouter quelques « règles de bonnes pratiques » supplémentaires. Certaines de ces règles peuvent être assouplies pour les programmeurs avertis qui n'ont plus besoin de « guide » pour bien coder.

Ces règles obligent les codes à respecter l'ordre « normal » d'exécution. Par exemple, si le code contient une instruction « if », « while » ou « for » ou une séquence d'instructions, il nous semble important pour bien comprendre la logique du code, de ne pas interrompre ce code en plein milieu.

Malheureusement la plupart des langages de programmation dits « impératifs » comme Python permet de déroger à cette règle en offrant au programmeur certaines instructions de « rupture de séquence ». Nous ne vous en avons pas parlé puisque nous ne voulons pas que vous les utilisiez ! *Nous comparons souvent l'utilisation de ce type d'instructions, à quelqu'un qui visite une maison et qui en sort par la fenêtre : ce n'est pas très poli !*

En fait au sein des fonctions, nous vous avons parlé d'une de ces instructions : l'instruction `return`. Elle permet de sortir d'une instance de fonction pour revenir au code appelant. Le fonctionnement de Python oblige de terminer l'exécution de toute fonction par un `return` explicite si l'on veut renvoyer une valeur précise, ou implicite si la fonction renvoie la valeur `None`. Il n'est donc pas question de proscrire l'instruction `return`, mais bien de la cadrer de façon précise.

La règle de bonne pratique que nous voulons que vous respectiez est la suivante :

- Une fonction de type `None` ne doit pas contenir de `return`.
- Une fonction d'un type différent de `None` ne doit contenir qu'un seul `return` avec le résultat, et ce `return` doit être placé comme dernière instruction de cette fonction en dehors de toute instruction composée (if, while, for, ...).

Habituez-vous à respecter cette règle qui vous permet d'écrire du code plus « propre » : ce n'est pas si difficile !

Et pourquoi ne pas reprendre vos codes UpyLaB précédents et s'ils ne suivent pas les règles de bonnes pratiques de codage, les corriger ?

LE PEP 20 EN MODE MONTY PYTHON (PARTIE OPTIONNELLE)

Parmi les recommandations et améliorations pour le langage Python, répertoriées dans les PEPs (Python Enhancement Proposals), nous trouvons le PEP 20. Le PEP 20 résume les 20 aphorismes (dont seulement 19 ont été rédigés) utilisés par Guido van Rossum, le BDFL Python (Benevolent Dictator for Life) jusqu'en 2018, pour concevoir le langage lui-même.

(Note : Le dictionnaire Larousse définit aphorisme par : phrase, sentence qui résume en quelques mots une vérité fondamentale. (Exemple : Rien n'est beau que le vrai.))

La lecture du PEP 20 est assez hermétique pour les programmeurs débutants ; ne vous arrêtez donc pas à cela. Si vous êtes quand même curieux de son contenu (en anglais) ouvrez une console PyCharm et tapez la commande :

```
import this
```

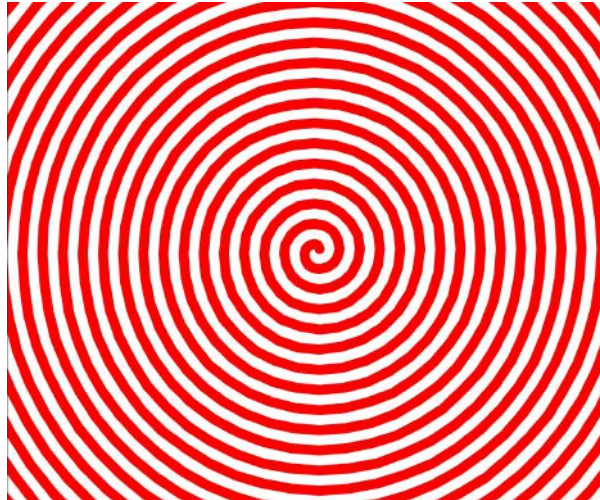
qui vous dévoilera son contenu inspiré par l'humour absurde des *Monty Python* dont Guido van Rossum était fêru.

4.5 Pratique des fonctions

4.5.1 Mettons ensemble en pratique ce que nous venons de voir

SPIRALE AVEC TURTLE

Comme exercice, nous vous demandons de dessiner une spirale ou une courbe assez proche d'une spirale avec le module turtle. La technique que nous vous proposons pour dessiner la spirale est assez simple : il suffit d'assembler bout à bout des quart de cercles de rayon de plus en plus grand.



La vidéo suivante montre l'affichage d'un petit programme qui appelle une fonction `spirale`. Le code utilise turtle et la fonction `circle` pour dessiner des quarts de tour de plus en plus grands. Les autres fonctions du module turtle utilisées par mon code sont `reset`, `color`, `speed` et `width`.

VIDÉO DE LA SPIRALE AVEC TURTLE

Note : Voir la vidéo de la section 4.5.1 : Exemple de dessin de spirale

EXERCICE À FAIRE ENSEMBLE

L'exercice suivant met en pratique ce que nous avons vu sur les fonctions. Écrivez, dans un script PyCharm, un code qui réalise un affichage similaire grâce à une fonction `spirale` qui a deux paramètres :

- la couleur,
- la largeur du trait.

Ainsi, la figure précédente est l'exécution répétée de : `spirale('red', 10)`

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir un autre code qui solutionne l'exercice demandé ? Nous vous proposons une solution.

```
""" auteur: Thierry Massart
    date : 10 avril 2018
    but du programme : trace avec turtle des spirales
"""

import turtle

def spirale(couleur, largeur):
    """ Dessine avec turtle une spirale
```

(suite sur la page suivante)

(suite de la page précédente)

```
paramètres :  
- couleur : la couleur à utiliser  
- largeur : la largeur du tracé"""  
  
turtle.color(couleur)  
turtle.width(largeur)  
for i in range(100):  
    turtle.circle(i*(largeur/2), 90)  
  
while True:  
    turtle.reset()  
    turtle.speed(0)  
    spirale('red', 10)
```

Notez que la solution proposée boucle indéfiniment : le programme n'arrêtera son exécution que lorsque l'on le « tue », par exemple avec la touche clavier cmd-F2 (commande-F2) ou CTRL-c (contrôle-c) en étant dans la fenêtre Run du script PyCharm en exécution.

4.5.2 Autre exercice accompagné

POLYGONES AVEC TURTLE

Continuons de pratiquer la matière de ce module avec turtle qui nous permet de facilement visualiser le résultat et donc d'en valider notre compréhension. Supposons que nous désirions dessiner diverses figures. Nous pouvons, pour ce faire, imaginer définir différentes fonctions. Prenons le cas d'une fonction qui dessine un polygone régulier.

À nouveau, la première chose à déterminer est le nom de la fonction et quels paramètres elle aura.

Supposons que l'on appelle cette fonction `polygone_turtle`. Il faut ensuite penser à l'ensemble des paramètres.

Les paramètres pourraient être :

- le nombre de côtés du polygone régulier ;
- les coordonnées (x, y) du centre de la figure sur la fenêtre turtle ;
- la taille du rayon du cercle qui circonscrit le polygone ;
- la couleur du polygone.

Par exemple, le dessin suivant provient d'un programme qui a appelé la fonction `polygone_turtle` et défini cinq couleurs différentes, le nombre de côtés, le centre et le rayon étant choisis aléatoirement.

RÉSULTAT DU PROGRAMME POLYGONE_TURTLE.PY

Le code dessine séquentiellement 5 polygones remplis : noir, bleu, rouge, vert jaune de taille, nombre de côtés et emplacement choisis pseudo-aléatoirement.

EXERCICE À RÉALISER

À vous !

Écrivez dans un script PyCharm un programme complet `figures_geometriques` qui définit et utilise cette fonction `polygone_turtle`. Vous pouvez également le compléter d'une fonction `etoile_turtle(n, x, y, rayon, couleur)` qui dessine une étoile à n branches avec les mêmes spécifications qu'avec la fonction `polygone_turtle`.

Ici nous ne vous donnons pas une solution complète. Comme le résultat est graphique, il vous sera possible de vérifier s'il fonctionne correctement. Nous vous donnons quand même une aide pour réaliser le programme `figures_geometriques`, si vous avez des soucis avec les formules trigonométriques.

Petite aide pour réaliser le programme `figures_geometriques`

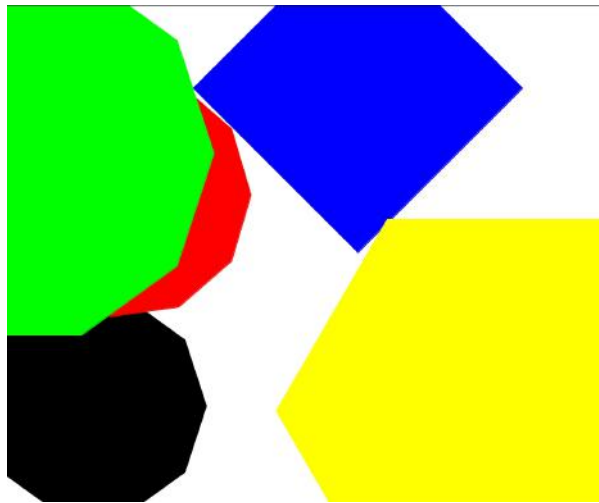


Fig. 4.4 – Exemple de dessin de polygones

Ayant importé `turtle`, `cos`, `sin` et `pi`, après avoir positionné, avec `turtle.goto(x+rayon, y)` la tortue au premier point du polygone de centre (x, y) et de rayon `rayon`, le code suivant trace le polygone à `n` côtés

```
for i in range(1, n + 1):
    turtle.goto(x + rayon * cos(i * 2 * pi / n), y + rayon * sin(i * 2 * pi / n))
```

4.5.3 Le yin et le yang revisité et paramètres par défaut

YIN-YANG REVISITÉ

Reprenons le code de la fonction `yin_yang` donné en section 4.3 pour le corriger en suivant les bonnes pratiques discutées avant dans ce module. Cette fonction avait été proposée sans paramètre. Pour respecter les règles de bonnes pratiques d'un codeur Python, cette fonction a plusieurs défauts :

1. elle est trop longue en terme de séquence monolithique de lignes. Il faudrait essayer de la découper en utilisant plusieurs sous-fonctions.
2. elle devrait avoir des paramètres :
 - ici le yin-yang a un rayon de 200 points. Si nous voulons faire un yin-yang d'une autre taille, il faudrait corriger des valeurs un peu partout dans le code, ce qui peut être une belle source d'erreurs.
 - on pourrait imaginer un yin-yang avec d'autres couleurs, non centré, ...
3. si l'on regarde le code, il fait plus ou moins deux fois la même chose puisque le yin a la même forme que le yang. Seuls l'emplacement et la couleur changent. Il serait donc bien d'essayer de n'avoir qu'un seul code qui trace à la fois le yin et le yang en fonction des arguments donnés à l'exécution.

PARAMÈTRES AVEC VALEUR PAR DÉFAUT

Python propose de donner des valeurs par défaut aux paramètres lors de sa définition. Rappelons que « par défaut » signifie si rien n'est donné explicitement. Ici, par défaut signifie donc « si lors de l'appel à la fonction la valeur de l'argument n'est pas donnée ».

Par exemple nous pouvons définir l'entête d'une fonction `yin_yang` comme suit : `def yin_yang(rayon, color1='black', color2='white')` : le corps de la fonction étant défini normalement. Avec cet entête, si lors de l'appel à la fonction `yin_yang`, seulement deux paramètres sont donnés, le premier argument sera donné pour le paramètre `rayon` et le second pour le paramètre `color1`. Le paramètre `color2` prendra la valeur par défaut `'white'`.

Si lors d'un appel à la fonction `yin_yang`, un seul argument est donné (pour le paramètre `rayon`), `color1` vaudra sa valeur par défaut `'black'` et `color2` sa valeur par défaut c'est-à-dire `'white'`. Notons que comme le paramètre `rayon` n'a pas de valeur par défaut dans sa définition, il faudra au moins donner un argument lors de tout appel à la fonction.

Notons que Python a également la possibilité d'effectuer la transmission des arguments aux paramètres par mot-clé comme avec `print(res1, res2, res3, end="", sep="/")` qui spécifie que les paramètres `end` et `sep` prennent respectivement la valeur chaîne de caractères vide (`""`) et la caractères barre oblique.

À vous de jouer !

Nous vous proposons donc, à nouveau dans PyCharm, de modifier le code précédent pour mieux le structurer et permettre d'autres couleurs (le contour restant en noir).

Pour vous aider, voici la structure proposée : la fonction `yin_yang` définit en son sein deux fonctions :

- la fonction `yin` qui fait la moitié du symbole (soit le yin soit le yang). Elle sera donc appelée deux fois par la fonction `yin_yang`;
- la fonction `yang` qui fait le disque yang (respectivement yin) à l'intérieur du yin (resp. yang) et qui est appelée au sein de la fonction `yin`.

L'ossature du code proposé est donc comme suit :

OSSATURE CODE PYTHON

```
""" code avec fonction yin_yang dessiné avec turtle """
import turtle

def yin_yang(rayon, color1='black', color2='white'):
    """ dessine un logo yin-yang de rayon rayon """

    def yang (rayon, couleur1, couleur2):
        """ dessin du yang à l'intérieur du yin (ou vice versa) """
        pass # code TODO

    def yin(rayon, couleur1, couleur2):
        """ dessine la moitié d'un yin-yang
            utilise la fonction yang """
        pass # code TODO

    #code de yin_yang
    turtle.reset()
    turtle.width(2)
    yin(rayon, color1, color2)
    yin(rayon, color2, color1)
    turtle.hideturtle()
    return

#code principal
yin_yang(200) #réalise le logo de rayon 200
```

Si vous ne trouvez pas la bonne réponse, n'hésitez pas à en discuter via le [Forum général du Module 4] pour y arriver !

Si malgré tout, vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution, nous vous en proposons une.

PROPOSITION DE SOLUTION

Ci-dessous une proposition de solution.

```

""" auteur : Thierry Massart
    date : 10 avril 2018
    code avec fonction yin_yang dessiné avec turtle """
import turtle

def yin_yang(rayon, color1='black', color2='white'):
    """ dessine un logo yin-yang de rayon rayon """

    def yang (rayon, couleur1, couleur2):
        """ dessin du yang à l'intérieur du yin """
        turtle.left(90)
        turtle.up()
        turtle.forward(rayon*0.35)
        turtle.right(90)
        turtle.down()
        turtle.color(couleur1, couleur2)
        turtle.begin_fill()
        turtle.circle(rayon*0.12)
        turtle.end_fill()
        turtle.left(90)
        turtle.up()
        turtle.backward(rayon*0.35)
        turtle.down()
        turtle.left(90)

    def yin(rayon, couleur1, couleur2):
        """ dessine la moitié d'un yin-yang le contour étant en noir"""
        turtle.color("black", couleur1)
        turtle.begin_fill()
        turtle.circle(rayon/2., 180)      # demi cercle de rayon / 2
        turtle.circle(rayon, 180)        # demi cercle de rayon
        turtle.left(180)
        turtle.circle(-rayon/2., 180)    # demi cercle intérieur de rayon / 2
        turtle.end_fill()
        yang(rayon, couleur1, couleur2) # dessine le yang à l'intérieur du yin

    #code de yin_yang
    turtle.reset()
    turtle.width(2)
    yin(rayon, color1, color2)
    yin(rayon, color2, color1)
    turtle.hideturtle()
    return

#code principal
yin_yang(200) #réalise le logo

```

4.5.4 Un pavage du projet Vasarely avant déformation avec fonctions

ÉNONCÉ DE L'EXERCICE

Pour le projet Vasarely proposé au module suivant, nous avons écrit durant l'activité 2.6.3 du code pour tracer un pavé. L'activité suivante va construire quelques briques de plus pour le projet. Pour cela, nous vous demandons de reprendre le code que vous avez produit en 2.6.3, et d'en faire une fonction :

```
pave(abscisse_centre, ordonnee_centre, longueur_arete, color1, color2, color3)
```

qui trace dans une fenêtre turtle

- à partir du point centre donné par les coordonnées (`abscisse_centre`, `ordonnee_centre`)
- un pavé dont chaque arête à une longueur `longueur_arete`
- avec des 3 losanges remplis respectivement par les `color1`, `color2` et `color3`.

Pavé noir, bleu et rouge

Votre fonction déplacera d'abord turtle au point centre sans tracer (grâce à la fonction `turtle.up`) et ensuite tracera le pavé (fonction `turtle.down` et `turtle.goto`).

Conseils :

- Comme lors du composant 2.6.3, nous vous conseillons d'utiliser la fonction `goto` pour effectuer les tracés. Attention ici, si le pavé n'est pas au centre de la fenêtre, il faut tenir compte des coordonnées du centre lors des appels à `turtle.goto`.
- Pour simplifier, utilisez une version sans instruction `for` ou `while`.

Vous pourrez ensuite tester votre fonction `pave` avec un code l'appelant avec différentes valeurs d'arguments pour dessiner des pavés à différents endroits de la fenêtre turtle et de différentes tailles.

Par exemple vous pouvez compléter le code suivant, qui fait différents appels avec des valeurs aléatoires, en ajoutant votre code de la fonction `pave`.

CANEVAS DE CODE

```
import turtle
from math import pi, sin, cos
import random
import time

def pave(abscisse_centre, ordonnee_centre, longueur_arete, color1, color2, color3):
    """ Dessine avec turtle un pave hexagonal
        en position ( abscisse_centre, ordonnee_centre)
        paramètres :
        - (abscisse_centre, ordonnee_centre) : point centre du pavé
        - longueur_arete : longueur de chaque arête du pavé
        - color1, color2, color3 : les couleurs des 3 hexagones"""
    # TODO code de fonction pave ici

turtle.hideturtle()
turtle.speed(0)
turtle.reset()
time.sleep(5)

while True:
    pave(random.randint(-300,300), random.randint(-300,300),
         random.randint(10,50), 'black', 'red', 'blue')
    pave(random.randint(-300,300), random.randint(-300,300),
         random.randint(10,50), 'white', 'grey', 'black')
```

Votre code complet devrait produire un résultat similaire à celui de la vidéo de la section 4.5.5

RÉSULTAT DU CODE

Note : Voir la vidéo de la section 4.5.5 : Dessins de pavés hexagonaux

Note : Notez que la seconde partie de la vidéo est la séquence vidéo inversée réalisée avec le logiciel screenflow.

4.5.5 Mise en pratique : exercice UpyLaB 4.5

MISE EN PRATIQUE AVEC UPYLAB

Nous sommes arrivés à la fin de ce module, où vous devez devenir autonome dans la rédaction de code Python utilisant des fonctions. Pour cela réalisez tous les exercices UpyLaB du module 4 qu'il vous reste à faire.

N'oubliez pas qu'UpyLaB est un environnement de test et non un environnement de développement. En particulier, la plupart des codes demandés par la suite par UpyLaB ne sont qu'une partie d'un programme complet. Par exemple, UpyLaB demande de définir une fonction `foo`, et complètera votre définition par un code qui testera l'exécution de votre fonction pour plusieurs jeux de paramètres. Avant de soumettre à UpyLaB, il est fortement conseillé de développer dans l'environnement de développement PyCharm un programme complet qui définisse `foo` mais également qui fasse des appels pour tester si elle est correcte. C'est ce que l'on appelle réaliser des test unitaires ; l'unité étant un bout de code comme une fonction, ici la fonction `foo`.

Bon travail !

EXERCICE UPYLAB 4.5 (Parcours Vert, Bleu et Rouge)

Énoncé

Écrire une fonction `alea_dice(s)` qui génère trois nombres (pseudo) aléatoires à l'aide de la fonction `randint` du module `random`, représentant trois dés (à six faces avec les valeurs de 1 à 6), et qui renvoie la valeur booléenne `True` si les dés forment un 421, et la valeur booléenne `False` sinon.

Le paramètre `s` de la fonction est un nombre entier, qui sera passé en argument à la fonction `random.seed` au début du code de la fonction. Cela permettra de générer la même suite de nombres aléatoires à chaque appel de la fonction, et ainsi de pouvoir tester son fonctionnement.

Exemple 1

L'appel suivant de la fonction :

```
alea_dice(1)
```

doit retourner :

```
False
```

Exemple 2

L'appel suivant de la fonction :

```
alea_dice(25)
```

doit retourner :

```
True
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire seulement la fonction `alea_dice`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, et ne fait en particulier aucun appel à `input` ou à `print`.
- Il n'est pas demandé que la fonction `alea_dice` teste le type du paramètre reçu.

- N'importe quelle combinaison de trois dés permettant de former le nombre 421 sera acceptée, quel que soit l'ordre de la combinaison. Par exemple, les tirages 2, 4, 1 forment bien 421.
- La première instruction de la fonction, après l'import du module `random`, sera `random.seed(s)`.
- On rappelle que la fonction `randint(a, b)` retourne un nombre (pseudo) aléatoire compris entre les bornes `a` et `b` incluses.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Veillez à ce que la fonction retourne bien les valeurs booléennes `True` ou `False` et non les chaînes de caractères `"True"` ou `"False"`.
- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans PyCharm, pensez à ajouter l'instruction `print(alea_dice(s))`, en remplaçant `s` par une valeur entière. En particulier, vous pouvez remplacer ce paramètre `s` par l'argument utilisé par UpyLaB dans chacun de ces tests.
- [Python Tutor](#) pourra s'avérer particulièrement utile ici pour observer ce qu'il se passe exactement lors de l'appel de la fonction.
- Si rien ne marche : consultez la [FAQ sur UpyLaB 4.5](#).

Note pour réaliser les exercices qui suivent

Certains exercices dont l'exercice UpyLaB suivant utilisent des valeurs de type tuple. Les tuples sont des séquences de données qui seront étudiées en détail au module suivant. Pour réaliser les exercices de ce présent module, ayant des valeurs `v1` et `v2` (par exemple `int` ou `float`), il faut juste savoir que :

- `tuple()` : renvoie un tuple vide
- `(v1,)` : renvoie un tuple avec une composante de valeur `v1`
- `(v1, v2)` : renvoie un tuple avec les deux valeurs `v1` et `v2`.
- ...

Ainsi `(2, 3, 5, 7, 11)` est un 5-tuple (contient 5 valeurs).

4.5.6 Exercice UpyLaB 4.6 (Parcours Vert, Bleu et Rouge)

Énoncé

Considérons les billets et pièces de valeurs suivantes : 20 euros, 10 euros, 5 euros, 2 euros et 1 euro.

Écrire une fonction `rendre_monnaie` qui reçoit en paramètre un entier `prix` et cinq valeurs entières `x20`, `x10`, `x5`, `x2` et `x1`, qui représentent le nombre de billets ou de pièces de chaque valeur que donne un client pour l'achat d'un objet dont le prix est mentionné.

La fonction doit renvoyer cinq valeurs, représentant le nombre de billets et pièces de chaque sorte qu'il faut rendre au client, dans le même ordre que précédemment. Cette décomposition doit être faite en rendant le plus possible de billets et pièces de grosses valeurs.

Si la somme d'argent avancée par le client n'est pas suffisante pour effectuer l'achat, la fonction retournera cinq valeurs `None`.

Exemple 1

L'appel suivant de la fonction :

```
rendre_monnaie(38, 1, 1, 1, 1, 1)
```

doit retourner :

```
(0, 0, 0, 0, 0)
```

Exemple 2

L'appel suivant de la fonction :

```
rendre_monnaie(56, 5, 0, 0, 0, 0)
```

doit retourner :

```
(2, 0, 0, 2, 0)
```

Exemple 3

L'appel suivant de la fonction :

```
rendre_monnaie(80, 2, 2, 2, 3, 3)
```

doit retourner :

```
(None, None, None, None, None)
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire seulement la fonction `rendre_monnaie`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, et ne fait en particulier aucun appel à `input` ou à `print`.
- Il n'est pas demandé que la fonction `rendre_monnaie` teste le type des paramètres reçus.
- On suppose qu'il y a toujours suffisamment de billets et pièces de chaque sorte.
- Pour retourner cinq valeurs, on pourra utiliser l'instruction `return res20, res10, res5, res2, res1`.

Cela renvoie en réalité un tuple de cinq valeurs (apparaissant entre parenthèses lorsqu'on l'affiche). La notion de tuple sera introduite au module suivant ; pour l'instant, ne vous préoccupez pas de ceci.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans PyCharm, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(rendre_monnaie(70, 2, 1, 2, 2, 2))` par exemple.
- Pour déterminer le nombre de billets et pièces de chaque sorte, l'opérateur `//` pourra s'avérer utile, ainsi que l'opérateur `%` pour calculer la somme restant à décomposer.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.6*.

4.5.7 Exercice UpyLaB 4.7 (Parcours Vert, Bleu et Rouge)

Énoncé

Dans cet exercice, nous allons mettre en pratique la notion de valeur par défaut des paramètres d'une fonction.

Écrire une fonction `somme(a, b)` qui retourne la somme de deux valeurs entières `a` et `b`. Par défaut, la valeur de `a` est 0 et la valeur de `b` est 1.

Exemple 1

L'appel suivant de la fonction :

```
somme(24, 18)
```

doit retourner :

```
42
```

Exemple 2

L'appel suivant de la fonction :

```
sum(4)
```

doit retourner :

```
5
```

Exemple 3

L'appel suivant de la fonction :

```
sum()
```

doit retourner :

```
1
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire seulement la fonction `somme`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, et ne fait en particulier aucun appel à `input` ou à `print`.
- Il n'est pas demandé que la fonction `somme` teste le type des paramètres reçus.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans PyCharm, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(somme(3, 2))` par exemple.
- N'hésitez pas à revoir comment on a défini des valeurs par défaut pour certains paramètres de la fonction `yin_yang`.
- Si rien ne marche : consultez la [FAQ sur UpyLaB 4.7](#).

4.5.8 Exercice UpyLaB 4.8 (Parcours Bleu et Rouge)

Énoncé

Écrire une fonction `rac_eq_2nd_deg(a, b, c)` qui reçoit trois paramètres de type float correspondant aux trois coefficients de l'équation du second degré $ax^2 + bx + c = 0$, et qui renvoie la ou les solutions s'il y en a, sous forme d'un tuple.

Exemple 1

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0, -4.0, 4.0)
```

doit retourner :

```
(2.0,)
```

Exemple 2

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0, 1.0, -2.0)
```

doit retourner :

```
(-2.0, 1.0)
```

Exemple 3

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0, 1.0, 1.0)
```

doit retourner :

```
()
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire seulement la fonction `rac_eq_2nd_deg`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, et ne fait en particulier aucun appel à `input` ou à `print`.
- Il n'est pas demandé que la fonction `rac_eq_2nd_deg` teste le type des paramètres reçus.
- Le résultat retourné par la fonction `rac_eq_2nd_deg` est un tuple.
 - S'il n'y a pas de solution réelle, elle retourne un tuple vide `tuple()`.
 - S'il y a une unique racine `r1`, elle retourne le tuple `(r1,)`.
 - S'il y a deux solutions réelles, `r1` et `r2`, la plus petite des deux devra être la première composante du tuple retourné (composante d'indice 0). La fonction pourra retourner le tuple `(min(r1, r2), max(r1, r2))`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans PyCharm, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(rac_eq_2nd_deg(1.0, 1.0, 1.0))` par exemple.
- Les calculs permettant d'obtenir les éventuelles solutions de l'équation ont été présentés au module 3 de ce cours, lors de l'introduction de l'instruction conditionnelle `if`.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.8*.

4.5.9 Exercice UpyLaB 4.9 (Parcours Bleu et Rouge)

Énoncé

De Wikipedia (5 février 2019) :

En mathématiques, et plus particulièrement en combinatoire, les **nombre de Catalan** forment une suite d'entiers naturels utilisée dans divers problèmes de dénombrement.

Ils sont nommés ainsi en l'honneur du mathématicien belge Eugène Charles Catalan (1814-1894).

Les dix premiers nombres de Catalan (pour n de 0 à 9) sont :

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862.

Le nombre de Catalan d'indice n , appelé n -ième nombre de Catalan, est défini par :

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

où $n!$ désigne la factorielle de la valeur entière n :

$$n! = n(n-1)(n-2)\dots 1$$

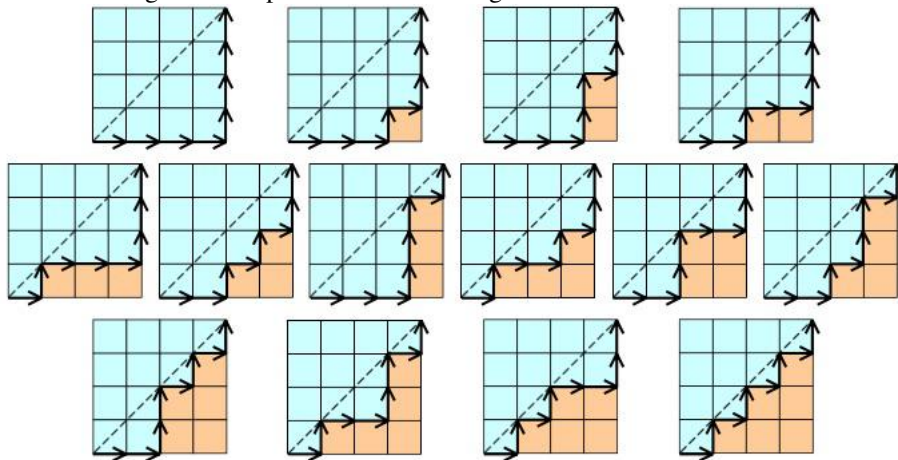
Par exemple, $5! = 5.4.3.2.1 = 120$ et

$$C_4 = \frac{8!}{5!4!} = 14$$

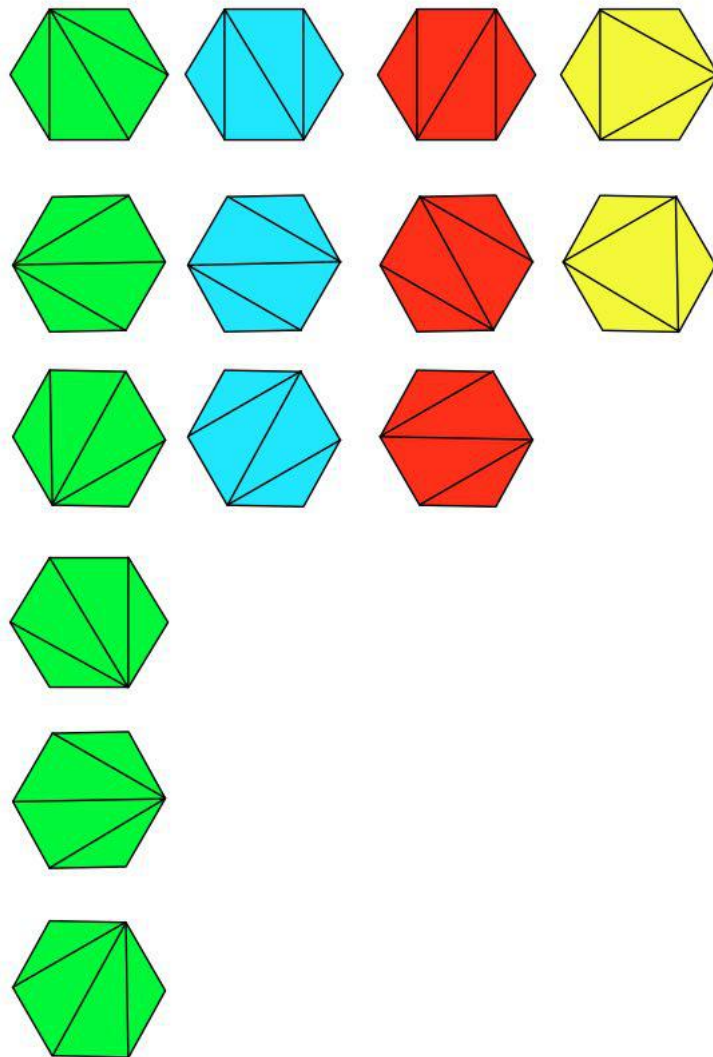
Le nombre de chemins sous-diagonaux les plus courts dans une grille de taille $n \times n$, le nombre de façons de découper en triangles un polygone convexe à $n+2$ côtés, ou encore le nombre de configurations possibles d'expressions avec n paires de parenthèses, appelé également mot de Dyck de longueur $2n$, sont des exemples dont le résultat est donné par le nombre de Catalan C_n .

Exemples d'applications de C_n (ici, $n = 4$)

— Nombre de chemins sous-diagonaux les plus courts dans une grille de taille $n \times n$



— Nombre de façons de découper en triangles un polygone convexe de taille $n+2$



— Nombre de parenthésages possibles (mots de Dyck)

((((()))	(((()))	((()))	((()))	()(())	(()))	(())()
()(())	()()()	(())()	(())()	()()()	(())()	(())()

Écrire une fonction `catalan(n)`, où `n` est un nombre entier positif ou nul, qui renvoie la valeur du `n`-ième nombre de Catalan.

Exemple 1

L'appel suivant de la fonction :

```
catalan(5)
```

doit retourner :

```
42
```

Exemple 2

L'appel suivant de la fonction :

```
catalan(0)
```

doit retourner :

```
1
```

Consignes

- Dans cet exercice, il ne vous est demandé que d'écrire une fonction. Votre code ne comportera donc aucun appel aux fonctions `input` et `print`.
- Il n'est pas demandé que la fonction `catalan` teste le type du paramètre reçu.
- Rappelons aussi que si une fonction est demandée, UpyLaB va tester à la fois que les résultats envoyés par cette fonction sont corrects et qu'ils ont le bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Vous pouvez utiliser la fonction `factorial` du module `math`, ou écrire votre propre fonction `factorielle` ou encore ne pas calculer explicitement ces factorielles en étudiant comment l'on passe d'un nombre de Catalan au suivant.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.9*.

4.5.10 Exercice UpyLaB 4.10 (Parcours Bleu et Rouge)

Cet exercice et le suivant vous demandent de programmer le petit jeu appelé « Pierre-feuille-ciseaux » ou « Pierre-papier-ciseaux » (et qui porte encore d'autres noms comme indiqué dans la page [Pierre-papier-ciseaux sur Wikipedia](#) que nous utilisons pour rédiger l'énoncé ci-dessous).

Énoncé

Pierre-feuille-ciseaux est un jeu effectué avec les mains et qui oppose un ou plusieurs joueurs. Ici nous nous supposons qu'il y a deux joueurs : l'ordinateur et le joueur lui-même.

Déroulement du jeu

Les deux joueurs choisissent simultanément un des trois coups possibles en le symbolisant de la main :

- Poing fermé : Pierre ;
- Main ouverte, doigts collés les uns aux autres : Feuille ;
- Main avec pouce, annulaire et auriculaire fermé, index et majeur ouvert en forme de V : Ciseaux.

Résultat du jeu :

- La pierre bat les ciseaux (en les émoussant),
- les ciseaux battent la feuille (en la coupant),
- la feuille bat la pierre (en l'enveloppant).

Ainsi chaque coup bat un autre coup, fait match nul contre le deuxième (son homologue) et est battu par le troisième.

Écrire une fonction `bat(joueur_1, joueur_2)` où `joueur_1` et `joueur_2` ont chacun une valeur entière 0, 1 ou 2, qui encode ce que le joueur a fait comme coup (0 : PIERRE, 1 : FEUILLE, 2 : CISEAUX) qui renvoie un résultat booléen :

- vrai si `joueur_1` bat le `joueur_2` :
- faux si `joueur_2` bat `joueur_1` ou fait match nul contre lui.

Exemple 1

L'appel suivant de la fonction :

```
bat(0, 0)
```

doit retourner :

```
False
```

Exemple 2

L'appel suivant de la fonction :

```
bat(0, 1)
```

doit retourner :

```
False
```

Exemple 3

L'appel suivant de la fonction :

```
bat(2, 1)
```

doit retourner :

```
True
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire seulement la fonction `bat`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, éventuellement accompagnée des trois définitions des constantes `PIERRE`, `FEUILLE`, `CISEAUX`, et ne fait en particulier aucun appel à `input` ou à `print`.
- Il n'est pas demandé que la fonction `bat` teste le type ou les valeurs des paramètres reçus.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans PyCharm, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(bat(2, 2))` par exemple. Notez que comme chaque paramètre peut avoir 3 valeurs, la fonction `bat` doit gérer 9 cas possibles. N'hésitez pas à les tester tous avant de mettre votre code dans UpyLaB.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.10*.

4.5.11 Exercice UpyLaB 4.11 (Parcours Bleu et Rouge)

Énoncé

Pierre-feuille-ciseaux est un jeu effectué avec les mains et qui oppose un ou plusieurs joueurs.

Déroulement du jeu

Les deux joueurs choisissent simultanément un des trois coups possibles en le symbolisant de la main :

- Poing fermé : Pierre;
- Main ouverte, doigts collés les uns aux autres : Feuille;
- Main avec pouce, annulaire et auriculaire fermé, index et majeur ouvert en forme de V : Ciseaux.

De façon générale, la pierre bat les ciseaux (en les émoussant), les ciseaux battent la feuille (en la coupant), la feuille bat la pierre (en l'enveloppant). Ainsi chaque coup bat un autre coup, fait match nul contre le deuxième (son homologue) et est battu par le troisième.

Écrire un programme qui réalise 5 manches du jeu Pierre-feuille-ciseaux entre l'ordinateur et le joueur. Chaque manche va consister en :

- la génération (pseudo) aléatoire d'un nombre entre 0 et 2 compris, à l'aide de la fonction `randint` du module `random`, et qui va représenter le coup de l'ordinateur (0 valant Pierre, 1 Feuille et 2 Ciseaux);
- la lecture en entrée (`input`) d'une valeur entière entre 0 et 2 compris qui représente le coup du joueur;
- l'affichage du résultat sous une des formes :
 - `coup_o bat coup_j : points`
 - `coup_o est battu par coup_j : points`
 - `coup_o annule coup_j : points`

où

- `coup_o` et `coup_j` sont respectivement le coup de l'ordinateur et du joueur : "Pierre" s'il a joué 0, "Feuille" s'il a joué 1 et "Ciseaux" s'il a joué 2.
- `points` donne le résultat des manches jusqu'à présent sachant que le compteur `points` part de zéro, et est incrémenté de un chaque fois que le joueur gagne une manche, et décrémenté de un chaque fois que l'ordinateur gagne une manche (les matchs nuls ne modifiant pas le compteur `points`).

À la fin des cinq manches, votre programme affichera Perdu, Nul ou Gagné suivant que le compteur est négatif, nul ou strictement positif.

Pour plus de clarté dans votre code, nous vous conseillons de définir les trois constantes symboliques :

```
PIERRE = 0
FEUILLE = 1
CISEAUX = 2
```

Par ailleurs, votre code doit importer le module `random` et, **avant de commencer les manches**, pour permettre à UpyLaB de valider les résultats, doit d'abord lire une valeur entière `s` et appeler la fonction `random.seed(s)`. Vous devez donc intégrer le code suivant :

```
import random

PIERRE = 0
FEUILLE = 1
CISEAUX = 2

...

s = int(input())
random.seed(s)
```

Votre code fera donc un appel à `random.seed` suivi de cinq appelq à `random.randint`, un par manche. Aucun autre appel à une fonction de `random` ne pourra être effectué.

Vous pouvez bien sûr utiliser la fonction `bat` de l'exercice 4.9 mais nous vous conseillons vivement de définir aussi d'autres fonctions (par exemple , une fonction qui réalise une manche et imprime la ligne de message) pour structurer votre code.

Exemple 1

Sachant que le code suivant :

```
random.seed(65)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
1
1
1
2
0
```

L'exécution du code avec les entrées :

```
65
0
1
2
1
0
```

doit afficher :

```
Feuille bat Pierre : -1
Feuille annule Feuille : -1
Feuille est battu par Ciseaux : 0
Ciseaux bat Feuille : -1
Pierre annule Pierre : -1
Perdu
```

Exemple 2

Sachant que le code suivant :

```
random.seed(1515)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
2
1
0
2
2
```

L'exécution du code avec les entrées :

```
1515
0
1
2
1
0
```

doit afficher :

```
Ciseaux est battu par Pierre : 1
Feuille annule Feuille : 1
Pierre bat Ciseaux : 0
Ciseaux bat Feuille : -1
Ciseaux est battu par Pierre : 0
Nul
```

Exemple 3

Sachant que le code suivant :

```
random.seed(2001)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
2
0
1
0
0
```

L'exécution du code avec les entrées :

```
2001
0
1
2
1
0
```

doit afficher :

```
Ciseaux est battu par Pierre : 1
Pierre est battu par Feuille : 2
Feuille est battu par Ciseaux : 3
Pierre est battu par Feuille : 4
Pierre annule Pierre : 4
Gagné
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire un programme contenant des fonctions.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu en respectant majuscules et minuscules et en veillant à n'avoir qu'un espace entre les mots et signes et aucune espace supplémentaire en fin de ligne. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte supplémentaire dans ce qui est imprimé (`print(points)` et non `print("résultat :", points)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.11*.

4.6 Bilan du module

4.6.1 Qu'avons-nous vu dans ce module ?

BILAN EN BREF

Note : Voir la vidéo de la section 4.6.1 : Bilan du module 4

BILAN DU MODULE

Nous voici à la fin de ce module. Nous y avons vu principalement comment structurer notre code grâce à la définition et l'utilisation de fonctions. De façon plus précise, nous avons vu :

- que plusieurs fonctions prédéfinies bien utiles sont disponibles ;
- que la librairie standard regorge de modules qui peuvent être importés et sont remplis de fonctions prédéfinies bien utiles également ;
- que le programmeur peut également définir des fonctions Python ;
- que la définition et l'utilisation de fonctions sont essentielles pour écrire un « bon » code, c'est-à-dire un code « bien structuré » facilement « lisible » pour le programmeur et toute personne qui devrait consulter le code ;
- que le mécanisme de passage de paramètres permet d'« isoler » le code d'une fonction du code appelant ;
- qu'il existe toutes sortes de formes de fonctions qui renvoient un résultat ou dont le but est d'effectuer un autre type de traitement ;
- qu'un résultat d'une fonction peut être un tuple de valeurs, ce qui permet en pratique à la fonction de renvoyer plusieurs résultats lors d'un seul `return`.

Nous avons également illustré tous ces concepts avec des exemples et la réalisation d'exercices supervisés ou réalisés de façon autonomes avec UpyLaB.

Dans ce module et les modules précédents, nous avons appris à écrire des codes complets et modulaires. Les deux modules qui vont suivre vont compléter le tableau. Nous allons voir les séquences de données. Ces types de données vont nous permettre de résoudre des problèmes beaucoup plus ambitieux que ce que nous avons fait jusqu'à présent. En route donc vers l'infini et au-delà ! Enfin presque.

+, 36
=, 38
#croisillon, 42
%, 33
évaluation paresseuse, 74
évaluations de votre apprentissage, 8

affectation, 38
aide-mémoire, 52
and, 69
assert, 127
assignation, 38
assignation multiple, 60
associativité, 33, 70

Bonnes pratiques Python, 143

caractère d'échappement "\", 59
caractère de continuation "\", 142
Catalan, 156
chaîne de caractères, 35
chaîne de caractères multiligne, 59
code "propre", 139
commentaire, 41, 42
concaténation, 36
Conjecture de Syracuse, 89
console, 12
constante, 39
contrôle de type, 127

déboguer, 44
diagramme d'état, 128
diagramme d'état, 40
division en nombre flottant, 33
division entière, 33
docstring, 41
documentation Python, 28

elif, 65
else, 65
entrée, 40
espaces de discussion, 9
exponentiation, 33

expression, 32

f-strings, 142
False, 65
Fibonacci, 130
Fibonacci suite de, 96
float, 32, 58
fonction
 corps, entête, paramètre, argument, 123
 définition, appel, return, 121
fonction booléenne, 124
fonction prédéfinie, 120
for, 93
format, 141
forums, 7, 9

help, 52

identificateur, 72
if, 65
incrément, 60
indentation, 73
informatique, 28
input, 40
instruction conditionnelle, 65
instruction répétitive, 87, 93
int, 32, 58
interpréteur Python 3, 12
instance, 52

keyword, 72

len, 36
ligne de continuation, 142
Linux, 14
lois de De Morgan, 70

MacOS, 13
math, 48
mode interactif, 33
modulo, 33
mot-clé, 72

None, 124

not, 69

opérateur arithmétique

+, -, *, //, /, **, %, 33

opérateur de mise à jour, 60

opérateur logique, 69

opérateur relationnel

==, <=, >=, !=, <, >, 67

or, 69

paramètre

passage, effectif, formel, argument, 123

valeur par défaut, 147

parcours, 5

pass, 116

pavé hexagonal, 54

pep 20, 144

pep 8, 139

polymorphisme, 127

portée, 130

print, 40

priorité, 33, 70

PyCharm, 12

Python Tutor, 27, 40

racines d'une équation du second degré, 68

random, 75

sémantique, 39

script, 12

sortie, 40

syntaxe, 39

syntaxe de l'instruction for, 94

syntaxe de l'instruction if, 71

syntaxe de l'instruction while, 88

table de vérité des opérateurs logiques, 70

tester, 33

Traceback, 130

Trinket, 15

True, 65

turtle, 48, 52

type, 33, 52

type entier, 33

type flottant, 33

type fractionnaire, 33

Ubuntu, 14

UpyLaB, 4, 24, 26, 44

valeur, 32

variable, 37

locale, globale, 127

while, 87

Windows, 12

yin-yang, 124, 147



À propos du cours

Vous avez un ordinateur, désirez apprendre à coder et êtes totalement ou partiellement débutant dans le domaine; vous êtes étudiant, professeur ou simplement une personne qui sente l'envie ou le besoin d'apprendre la programmation de base; ce cours utilise Python 3 comme clé pour vous ouvrir la porte de cette connaissance informatique.

Ce cours est orienté vers la pratique, et propose un matériel abondant pour couvrir l'apprentissage de la programmation de base, d'une part en montrant et expliquant les concepts grâce à de nombreuses capsules vidéo courtes et des explications simples, et d'autre part en vous demandant de mettre ces concepts en pratique d'abord de façon guidée et ensuite autonome. Plusieurs quiz, un projet individuel, et de nombreux exercices à réaliser et validés automatiquement avec notre outil UpyLaB intégré au cours, vous permettent de polir et ensuite de valider votre apprentissage.

Format du cours

Le cours s'étale sur 9 semaines et propose 3 parcours d'apprentissage et correspond à un travail hebdomadaire de 6 à 12 heures avec un projet évalué par les pairs. Si vous ne pouvez y consacrer 6h par semaine un parcours à allure libre est possible, mais sans projet ni attestation finale délivrée par FUN.

Les enseignants

SÉBASTIEN HOARAU

Sébastien Hoarau est maître de conférences à l'Université de la Réunion (UR) où, depuis plus de 20 ans, il enseigne la programmation aux étudiants de première année scientifique. Sa pédagogie est axée sur la pratique : quiz en cours, projets, utilisation de plateformes ludiques.

THIERRY MASSART

Thierry Massart est professeur à l'Université Libre de Bruxelles (ULB) où, depuis plus de 25 ans, il enseigne la programmation principalement aux étudiants de Sciences Informatique et de l'école Polytechnique de l'ULB. Il leur propose une pédagogie active orientée vers la pratique.

ISABELLE POIRIER

Isabelle est professeur agrégé de mathématiques et enseigne celles-ci depuis plus de 15 ans en établissement secondaire dans le sud de la France. Autodidacte en programmation (en particulier grâce à sa participation à plusieurs MOOC), elle met à profit son sens de la pédagogie et sa propre expérience pour venir en aide aux apprenants sur le forum et améliorer l'approche didactique proposée au bénéfice d'un meilleur apprentissage pour tous.