Les bases de PYTHON

Table des matières

1 Introduction

PYTHON est un langage de programmation (1991) créé par Guido van Rossum au Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.

C'est un langage de programmation impérative (on décrit *comment* faire les choses pas à pas), structurée (de façon organisée) , orientée objet, de haut niveau (algorithme plus proche du langage *humain* que du langage *machine*).

Il présente les avantages suivants :

- sa syntaxe est simple et concise, proche du "langage algorithmique".
- moderne. Très largement répandu dans l'industrie, l'enseignement et la recherche, notamment, pour ses applications scientifiques. Une large communauté participe à son développement,
- puissant, muni de nombreuses bibliothèques de fonctions, permettant de s'adapter à tout type d'utilisation et de contexte,
- pratique pour travailler sur des objets mathématiques, assez proche du langage mathématique,
- gratuit, disponible sur la plupart des plateformes (Windows, Mac, Linux, ...).

2 IDLE Spyder

PYTHON est un langage interprété, ce qui signifie que le code est lu, traduit en code binaire et exécuté dans la foulée. Cela permet une grande souplesse (on peut décider de faire exécuter une nouvelle instruction rapidement) mais empêche certaines optimisations.

Contrairement à la technique de compilation qui consiste à traduire une seule fois le programme entier en langage machine avant l'exécution, l'interprétation consiste à lire le programme et l'exécuter ligne par ligne, au moment de l'exécution.

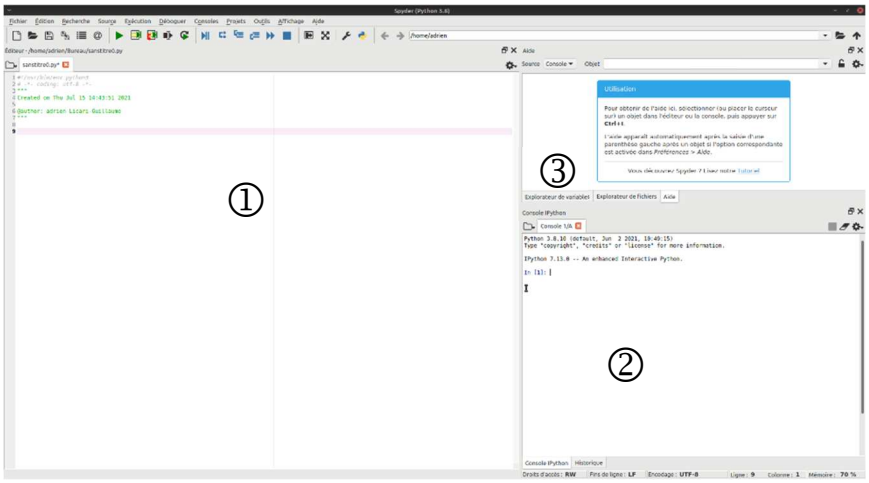
IDLE est l'environnement de développement et d'apprentissage intégré de Python (Integrated Development and Learning Environment), **interface graphique** qui aide à **écrire, exécuter et tester du code** facilement.

On peut distinguer 3 zones :

① L'éditeur (à gauche),

② L'interpréteur interactif (Console Ipython en bas à droite),

③ L'explorateur de variables, graphiques, aide (en haut à gauche).



2.1 L'interprète de commande, Shell ou Console

L'interprète de commande traduit les instructions Python en langage machine, il se trouve dans le Shell (la console). Dans l'environnement Spyder, l'interprète Ipython est automatiquement lancé, le Shell débute avec `In [1]:`. C'est une zone de dialogue interactif qui permet d'exécuter des commandes à la volée après le `In [1]:`.
Ipython numérote les instructions et distingue clairement une sollicitation (`In`) de sa réponse (`Out`).

```
In [1]: 1 + 1
Out[1]: 2
```

2.2 L'éditeur de texte ou script

Il permet de rédiger des programmes (script) que l'on exécute ensuite.
On utilise des instructions PYTHON, on respecte les règles d'indentation (présentées dans le paragraphe 5.1) , et pour faciliter la lecture, on utilise la convention PEP8 résumée en annexe 1 et on commente notre programme à l'aide du symbole #.
Les commentaires ne seront pas lus par le compilateur/interpréteur.

Par exemple :

```
1  print('1er calcul :')
2  print('2*3 =', 2 * 3)
3  # 2eme calcul
4  4 * 5
```

Après exécution, on lit dans la console :

```
In [2]: %runfile C:/Users/corin/.spyder-py3/script.py --wdir
1er calcul :
2*3 = 6
```

Lorsqu'on exécute un script écrit dans l'éditeur le résultat de ce dernier n'est pas retourné.
Les instructions `print` sont affichées, mais le résultat du calcul `2*3` n'est pas visible contrairement à une commande dans la console.
On utilisera la console pour tester tout ou partie d'une programme.

3 Variables, objets, typage

3.1 Notion de variable

Une variable permet de stocker des informations qui seront utilisées durant l'exécution d'un programme, elle est caractérisée par :

- un identificateur : nom de la variable explicite.
- un type : nature de la variable (booléen, entier, flottant, chaîne de caractères...).
- une valeur : l'affectation permet d'assigner la valeur d'un objet à une variable.
- une référence : la référence point directement vers l'adresse mémoire d'une variable.
- des opérations : combinaisons arithmétiques de deux ou plusieurs variables.

On parle de typage statique lorsqu'il est nécessaire de définir le type d'une variable lors de sa création et de typage dynamique lorsque le type le mieux adapté est choisi automatiquement lors de l'assignation d'une variable.

Au moment de l'exécution, PYTHON assigne le type adapté pour la variable sans qu'il ne soit déclaré, c'est un langage à typage dynamique.

Remarque : 33 noms de variables sont interdits dans le langage PYTHON

```
And      as      assert      break      class      continue      def      del      elif
else     except      False      finally      for      from      global      if
import      in      is      lambda      None      nonlocal      not      or      pass
raise      return      True      try      while      with      yield
```

3.2 Type des variables

3.2.1 Les numériques (pour les calculs numériques)

`int` : les nombres entiers	`In [3]: type(2)` `Out[3]: int`
`float` : les nombres décimaux (à virgule flottante)	`In [4]: type(2.0)` `Out[4]: float`
`Complex` : les nombres complexes	`In [5]: type(2 + 0j)` `Out[5]: complex`

3.2.2 Les booléens (pour les tests)

Les variables booléennes (type **`bool`**) sont des variables qui ne peuvent prendre que deux valeurs : `True` (vrai) et `False` (faux).
On peut simplement déclarer une variable de ce type : `proposition_1 = True`
Comme pour les nombres, PYTHON constate que `True` est un booléen, donc il en déduit que la variable `proposition_1` l'est également.

3.2.3 Les conteneurs

Les chaînes de caractères (succession de caractères de type string : `str`), les **listes** (collection de plusieurs éléments qui peuvent avoir un type différent et pouvant être modifiés : `list`), **les tuples** (collection de plusieurs éléments qui peuvent avoir un type différent et ne pouvant être modifiés : `tuple`), **les dictionnaires (collections** de clés auxquelles sont associées des valeurs : `dict`).

3.3 Affectation des variables

L'affectation consiste à lier un nom (variable) à une valeur (objet) à l'aide de l'opérateur =.
Une variable ne contient pas la *valeur*, elle pointe vers un objet.
Lors d'une affectation de valeur à une variable, un espace est réservé dans la mémoire de l'ordinateur. Cet espace mémoire est situé à une certaine adresse (référence).

3.3.1 Règle de nommage PEP8

Nom explicite formé d'une suite de lettres et de chiffres, qui doit commencer par une lettre, l'underscore _ est autorisé.

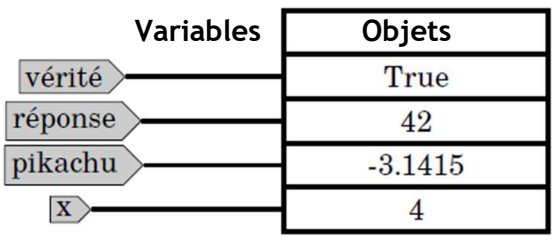
3.3.2 Fonctionnement de l'affectation

1. évaluer l'expression à droite de l'opérateur ;
2. inscrire la valeur en mémoire si elle ne l'est pas déjà ;
3. attacher la référence de gauche à la valeur.

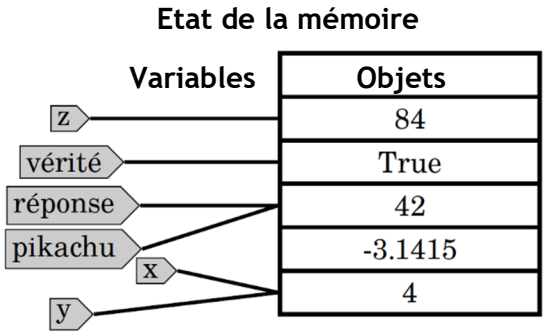
Exemple :

```
1 x = 4
2 pikachu = -3.1415
3 réponse = 42
4 vérité = (réponse == réponse)
```

Etat de la mémoire



```
5 y = x
6 pikachu = réponse
7 z = x * pikachu / 2
```



Voilà ce que fait PYTHON :

- ligne 5 : il **évalue** l'expression de droite, en lisant la valeur associée à la référence `x` : il en déduit que l'expression de droite vaut 4, valeur qui est déjà référencée en mémoire : il **attache** la référence `y` (créée à cette occasion) à cette valeur ;
- ligne 6 : il **évalue** l'expression de droite, en lisant la valeur associée à la référence réponse : il en déduit que l'expression de droite vaut 42, valeur qui est déjà référencée en mémoire : il **attache** la référence `pikachu` (simplement déplacée puisqu'elle existait déjà) à cette valeur ;
- ligne 7 : il **évalue** l'expression de droite, en lisant les valeurs associées aux références `pikachu` et `x` : il en déduit que l'expression de droite vaut 84, valeur qui n'est pas référencée en mémoire : il **inscrit en mémoire** cette valeur, puis il **attache** la référence `z` (créée à cette occasion) à cette valeur.

Remarques importantes :

- l'opérateur d'affectation n'est absolument pas symétrique (contrairement au signe = en mathématiques) : la partie de gauche et celle de droite ne sont pas du tout traitées de la même façon ;
- la partie de gauche doit être une **référence**, donc un **nom valide** ; la partie de droite peut être une expression, mais par exemple `a + b = 3` n'a aucun sens, puisque `a + b` n'est pas un nom mais une expression ;
- évaluer une variable qui n'a jamais été définie conduit à une erreur.
- les variables n'ont pas de type prédéfini, le type est automatiquement choisi par PYTHON en fonction de l'objet vers lequel la variable pointe. Sur l'exemple précédent `Pikachu` est de type `float` à la ligne 3 puis de type `int` à partir de la ligne 6.

3.3.3 Modification d'une variable à l'aide de sa propre valeur

On peut écrire des affectations qui « semblent fausses » sur le plan mathématique ; un exemple très courant est l'incrémentation (ou la décrémentation) : `nb_vies = nb_vies + 1`.
Si le signe = était une égalité, cela serait faux ; mais il s'agit d'une affectation :

1. on lit le nombre de vies (par exemple 4),
2. on ajoute 1 (dans l'exemple, cela donne 5),
3. puis on attache l'étiquette `nb_vies` à cette nouvelle valeur.

Ainsi, la variable `nb_vies` vaut 1 de plus après cette instruction qu'avant.

`a += b`	identique à `a = a + b`
`a -= b`	identique à `a = a - b`
`a *= b`	identique à `a = a * b`
`a /= b`	identique à `a = a / b`

3.3.4 Affectations multiples

Les expressions de droite sont **toutes évaluées** avant que les étiquettes ne soient affectées.

Par exemple :

```
a, b, c = 2, 3, 4
a, b = a**2, a*b # après cela, a vaut 2**2 -> 4 et b vaut 2*3 -> 6
```

3.4 Opérateurs

3.4.1 Opérateurs numériques sur les entiers et flottants

addition	soustraction	multiplication	division	division entière	reste division	puissance	valeur absolue	Incrémentation décrémentation
+	**-**	*****	**/**	**//**	**%** (modulo)	******	**abs**	**+=** **-=**

☞ Règles de priorité : Exposant-Modulo-Multiplication/Division-Addition/Soustraction (**EMoMDAS**)

Remarques :

- Des conversions de type de variable existent ; par exemple les entiers sont convertis en flottants lors d'une opération de division ou lors d'addition/multiplication entre entiers et flottants. Ainsi `10/10` donne `1.0` comme résultat.
- Certaines conversions sont possibles à l'aide des instructions : `int(2.3)` donne `2`, `float(1)` donne `1.0`.
- PYTHON convertit et stocke les flottants en binaire, valeur proche mais non exacte, ce qui pourra conduire à des résultats surprenants : `0.3 - 0.2` donne `0.09999999999999998`.
  La représentation des nombres fera l'objet d'un cours à part entière mais on retiendra dès à présent que l'**égalité entre nombres flottants n'a aucun sens.**

3.4.2 Opérateurs de comparaison

Interprétation	Pseudo-code	PYTHON
x est-il égal à y ?	$x = y$	`x == y`
x est-il distinct de y ?	$x \neq y$	`x != y`
x est-il strictement supérieur à y ?	$x > y$	`x > y`
x est-il supérieur ou égal à y ?	$x \geqslant y$	`x >= y`
x est-il strictement inférieur à y ?	$x < y$	`x < y`
x est-il inférieur ou égal à y ?	$x \leqslant y$	`x <= y`

Ne pas confondre l'opérateur de comparaison d'égalité `==` avec l'opérateur d'affectation `=` détaillé plus tard.

3.4.3 Opérateurs booléens

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

☞ Règles de priorité : `not` puis `and` puis `or`.

Remarque : les opérateurs **and** et **or** sont dits paresseux car dès que leur réponse peut être déterminée, la réponse est donnée sans terminer le calcul (inutile). Par exemple, dans l'expression `(a == b) or (a == c)`, si `a == b` est évaluée à `True`, alors `a == c` ne sera même pas évaluée, puisque, quel que soit le résultat de cette évaluation, l'ensemble sera nécessairement `True`.

L'utilisation privilégiée des booléens est le test, basé sur des opérations de comparaison vues plus haut.

4 Modules

Lorsqu'on utilise le langage PYTHON de base, seul un nombre très limité de fonctions est disponible ; mais l'utilisateur peut ajouter des modules additionnels en fonction de ses besoins et qui lui fournissent des fonctions supplémentaires.

Par exemple, les modules `math`, `numpy` ou `scipy` donnent accès :

- aux fonctions trigonométriques `sin`, `cos`, `tan` ainsi que leurs fonctions réciproques `arcsin`, `arccos`, `arctan` (les angles s'expriment par défaut en radians) ;
- aux fonctions exponentielle `exp`, logarithme népérien `log` ou logarithme décimal `log10` ;
- à la fonction racine carrée `sqrt`. Il en existe de nombreuses autres (voir l'aide `help`...).

Importation d'un module :

`import numpy`	pour calculer un sinus il faudra utiliser la fonction `numpy.sin`	Toutes les fonctions de ce module devront être préfixées du nom du module ou alias pour être utilisées
`import numpy as np`	Utilisation d'un alias : pour calculer un sinus il faudra utiliser la fonction `np.sin`	
`from numpy import *`	pour calculer un sinus il faudra utiliser la fonction `sin`	les fonctions n'ont pas besoin d'être préfixées par le nom du module ou alias
`from numpy import sin, cos`		

Remarques :

- Le mode d'importation `from numpy import *` est <u>à éviter</u> car si on utilise plusieurs modules différents qui contiennent des fonctions de même nom, c'est la fonction issue de la dernière importation qui sera utilisée. Par exemple les fonctions `cos` de `math` et de `numpy` n'utilisent pas le même type de variable :
  `scalaire` → `math.cos(scalaire)` alors que `tableau` → `numpy.cos(tableau)`
- D'autre part, ce mode d'importation présente un autre défaut : il charge en mémoire l'intégralité des définitions contenues dans ce module.

Modules classiquement utilisés et leur alias :

`math`	`m`	`import math as m`	calculs mathématiques élémentaires sur scalaires
`numpy`	`np`	`import numpy as np`	calcul numérique à l'aide de tableaux ou vecteurs
`scipy`	`sc`	`import scipy as sc`	calcul scientifique avancé
`matplotlib.pyplot`	`plt`	`import matplotlib.pyplot as plt`	représentation graphique

5 Fonctions

5.1 <u>Notion de structuration et indentation</u>

La programmation structurée nécessite la définition de blocs d'instructions au sein des structures de contrôles (`def, for, while, if`,...). Certains langages utilisent des délimiteurs pour encadrer ces blocs d'instructions (des parenthèses en C, des accolades en JAVA, des mots-clés en FORTRAN, etc), mais le langage PYTHON se distingue en utilisant l'**indentation**, qui favorise la lisibilité du code. Le début d'un bloc d'instructions est défini par un double-point (:), la première ligne pouvant être considérée comme un *en-tête*. Le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et le retour à l'indentation de l'en-tête marque la fin du bloc.

```
1  En-tête:
2      Bloc instructions . . .
3      . . . . . .
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres :

```
En-tête 1:
    Bloc 1 . . .
    . . . . . .
    En-tête 2:
        Bloc 2 . . .
        . . . . . .
    Suite Bloc 1
```

Cette structuration sert entre autre à définir de nouvelles fonctions, à réaliser des tests ou à effectuer des instructions répétitives.

5.2 Définition d'une fonction

On définit une fonction en PYTHON à l'aide du mot clé `def`. Il faut lui attribuer un nom, préciser la liste de ses paramètres et enfin décrire les différentes instructions à réaliser.
La syntaxe générale est la suivante :

```
Def nom_de_la_fonction(arguments):
    bloc instructions . . .
    return . . . . . .
```

De nouvelles variables peuvent être définies et utilisées dans le bloc d'instructions d'une fonction. De telles variables sont qualifiées de *locales* par opposition aux variables *globales*, dont le contenu est accessible à tout niveau, leur contenu est inaccessible depuis l'extérieur de la fonction et en particulier au niveau de l'interprète de commande.

Le principe de typage dynamique de PYTHON implique que le type est associé à la valeur (objet), pas à la variable, et est déterminé à l'exécution, pas à l'écriture du code.
On n'impose donc pas les types des arguments d'une fonction. Mais on peut les préciser à l'aide d'annotations de type qui facilitent la lisibilité.

```
Def addition(a: int, b: int) -> int:
    return a + b
```

Après exécution du script:

```
In [2]: addition(2, 3) # arguments entiers
Out[2]: 5
```

Mais:

```
In [3]: addition([1, 2],[4, 5, 6]) # listes [1, 2] et [4, 5, 6]
Out[3]: [1, 2, 4, 5, 6]
```

```
In [3]: addition('toto','tutu') # chaines de caractère 'toto' et 'tutu'
Out[3]: 'tototutu'
```

Remarque : on appelle procédure une fonction qui ne retourne pas de résultat et se contente d'agir sur l'environnement (sans `return`).

ANNEXE 1 : Convention et bonnes pratiques PEP8 (PYTHON Enhancement Proposal)

le PEP8 est un guide pour bien coder en PYTHON. Ce texte liste les règles stylistiques recommandées, invitant toute la communauté PYTHON à écrire un code de la même façon.

Les principales règles de PEP8 sont les suivantes : les lignes ne doivent pas dépasser 79 caractères ; l'indentation doit être uniforme et cohérente ; le code doit être organisé et documenté; les noms de variables et de fonctions doivent être clairs et informatifs ; les fichiers doivent être encodés en UTF-8 ; les commentaires doivent être concis et informatifs.

Catégorie	Règle	Bonne pratique ✓	Mauvaise pratique ✗
Indentation	Indentation standard	4 espaces	Tabulations
	Continuité de ligne	Indentation suspendue ou alignement vertical	Décalage incohérent
Longueur de ligne	Code	≤ 79 caractères	Lignes trop longues
	Commentaires / docstrings	≤ 72 caractères	Paragraphes larges
Espaces	Affectation	`x = 1` ✓	`x=1` ✗
	Opérateurs	`a + b, a == b`✓	`a+b, a==b` ✗
	Virgules	`a, b, c` ✓	`a,b,c` ✗
	Parenthèses	`func(a, b)` ✓	`func( a, b )` ✗
	Dictionnaires	`{"a": 1, "b": 2}` ✓	`{"a":1,"b":2}` ✗ ou `{"a" : 1 , "b" : 2}` ✗
Lignes vides	Fonctions / classes	2 lignes vides	Aucune séparation
	Méthodes	1 ligne vide	Bloc compact
Imports	Ordre	Standard → Tiers → Local	Imports mélangés
	Style	1 `import` par ligne	`import os, sys` ✗
	Position	Début de fichier	Imports au milieu
Nommage	Variables	`snake_case` ✓	`camelCase` ✗
	Fonctions	`compute_total()` ✓	`ComputeTotal()` ✗
	Classes	`CamelCase` ✓	`camel_case` ✗
	Constantes	`MAX_SIZE` ✓	`maxSize` ✗
Comparaisons	None	`if x is None:` ✓	`if x == None:` ✗
	Booléens	`if is_valid:` ✓	`if is_valid == True:` ✗
Conditions	Simplicité	`if items:` ✓	`if len(items) > 0:` ✗
Commentaires	Ligne	`# Commentaire clair`	`#commentaire`
Docstrings	Format	Triple guillemets `""" … """`	Commentaire simple
	Contenu	Quoi / Pourquoi	Comment
Chaînes de caractères	Guillemets	`'…'` ou `"…"` (cohérent)	Mélange

Fonctions trigonométriques

`math.` **acos**(*x*)
Return the arc cosine of *x*, in radians. The result is between `0` and `pi`.

`math.` **asin**(*x*)
Return the arc sine of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.` **atan**(*x*)
Return the arc tangent of *x*, in radians. The result is between `-pi/2` and `pi/2`.

`math.` **atan2**(*y*, *x*)
Renvoie `atan(y / x)`, en radians. Le résultat est entre `-pi` et `pi`. Le vecteur du plan allant de l'origine vers le point `(x, y)` forme cet angle avec l'axe X positif. L'intérêt de `atan2()` est que le signe des deux entrées est connu. Donc elle peut calculer le bon quadrant pour l'angle. par exemple `atan(1)` et `atan2(1, 1)` donnent tous deux `pi/4`, mais `atan2(-1, -1)` donne `-3*pi/4`.

`math.` **cos**(*x*)
Renvoie le cosinus de *x* radians.

`math.` **dist**(*p*, *q*)
Renvoie la distance Euclienne entre deux points *p* et *q*, passés comme des séquences (ou des itérables) de coordonnées. Les deux points doivent avoir la même dimension.

À peu près équivalent à :

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Nouveau dans la version 3.8.

`math.` **hypot**(**coordinates*)
Renvoie la norme Euclidienne, `sqrt(sum(x**2 for x in coordinates))`. C'est la norme du vecteur entre l'origine et le point donné par les coordonnées.

Pour un point bi-dimensionnel `(x, y)`, c'est équivalent à calculer la valeur de l'hypoténuse d'un triangle rectangle en utilisant le théorème de Pythagore, `sqrt(x*x + y*y)`.

Modifié dans la version 3.8: Ajout de la gestion des points à n-dimensions. Auparavant seuls les points bi-dimensionnels étaient gérés.

`math.` **sin**(*x*)
Renvoie le sinus de *x* radians.

`math.` **tan**(*x*)
Renvoie la tangente de *x* radians.

Conversion angulaire

`math.` **degrees**(*x*)
Convertit l'angle *x* de radians en degrés.

`math.` **radians**(*x*)
Convertit l'ange *x* de degrés en radians.

Fonctions logarithme et exponentielle

`math.` **exp**(*x*)
Renvoie *e* à la puissance *x*, où *e* = 2.718281… est la base des logarithmes naturels. Cela est en général plus précis que `math.e ** x` ou `pow(math.e, x)`.

`math.` **expm1**(*x*)
Renvoie *e* à la puissance *x*, moins 1. Ici, *e* est la base des logarithmes naturels. Pour de petits flottants *x*, la soustraction `exp(x) - 1` peut résulter en une perte significative de précision; la fonction `expm1()` fournit un moyen de calculer cette quantité en précision complète :

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Nouveau dans la version 3.2.

`math.` **log**(*x*[, *base*])
Avec un argument, renvoie le logarithme naturel de *x* (en base *e*).

Avec deux arguments, renvoie le logarithme de *x* en la *base* donnée, calculé par `log(x)/log(base)`.

`math.` **log1p**(*x*)
Renvoie le logarithme naturel de *1+x* (en base *e*). Le résultat est calculé par un moyen qui reste exact pour *x* proche de zéro.

`math.` **log2**(*x*)
Renvoie le logarithme en base 2 de *x*. C'est en général plus précis que `log(x, 2)`.

Nouveau dans la version 3.3.

Voir aussi: `int.bit_length()` renvoie le nombre de bits nécessaires pour représenter un entier en binaire, en excluant le signe et les zéros de début.

`math.` **log10**(*x*)
Renvoie le logarithme de *x* en base 10. C'est habituellement plus exact que `log(x, 10)`.

`math.` **pow**(*x*, *y*)
Renvoie `x` élevé à la puissance `y`. Les cas exceptionnels suivent l'annexe 'F' du standard C99 autant que possible. En particulier, `pow(1.0, x)` et `pow(x, 0.0)` renvoient toujours `1.0`, même si `x` est zéro ou NaN. Si à la fois `x` *et* `y` sont finis, `x` est négatif et `y` n'est pas entier, alors `pow(x, y)` est non défini et lève une `ValueError`.

À l'inverse de l'opérateur interne `**`, la fonction `math.pow()` convertit ses deux arguments en `float`. Utilisez `**` ou la primitive `pow()` pour calculer des puissances exactes d'entiers.

`math.` **sqrt**(*x*)
Renvoie la racine carrée de *x*.