Les messages d'erreur

Il existe deux grands types d'erreurs : les erreurs détectables lors de la traduction du source en programme et les erreurs apparaissant lors de l'exécution. Les premières correspondent généralement à des erreurs de syntaxe : faute de frappe, terme inconnu, structure incorrecte… Les secondes correspondent à des erreurs durant le déroulement du programme : tentative de division par zéro (c'est interdit), ouverture d'un fichier non existant ou encore utilisation d'un élément non initialisé...

Python, par la simplicité de son langage, permet de gagner beaucoup de temps de développement en réduisant les erreurs de syntaxe, ce qui reste très appréciable pour les développeurs débutants comme pour les plus confirmés.

Vous vous sentez fin prêt pour taper vos premières lignes de code ! Il faut néanmoins prendre le temps d'étudier la gestion des erreurs. En effet, depuis le plus jeune âge de l'informatique, les messages d'erreur sont peu compréhensibles. Ceci pour plusieurs raisons. D'une part, ils sont rédigés dans la langue de Shakespeare et il s'agit souvent de termes techniques que vous ne trouverez pas dans un dictionnaire. Ensuite, ils sont donnés dans un certain contexte, le contexte où l'interpréteur Python pensait être avant "le déraillement". Ainsi, si l'interpréteur pense qu'il est en train de lire une condition alors que ce n'est pas du tout le cas, vous obtiendrez un message d'erreur expliquant que votre condition est erronée alors qu'aucune condition ne se trouve écrite dans votre code à l'endroit désigné. Ce scénario se présente 60 % du temps et cette situation déroute tout programmeur, particulièrement les débutants, mais aussi les programmeurs expérimentés. Vous allez vous retrouver seul face à des messages compliqués, mieux vaut être préparé pour avoir quelques repères afin de sortir de l'impasse sans perdre trop de temps. La bonne nouvelle est que l'on retrouve souvent les mêmes messages et qu'à force on arrive à décrypter rapidement le problème sous-jacent.

Vous croiserez certaines personnes prétendant ne jamais faire d'erreur en programmant ou pour qui programmer sans faire d'erreur est chose aisée. Il n'y a rien à répondre à cela, tout le monde a le droit de rêver !

1. SyntaxError : invalid syntax

Les fautes de frappe, d'orthographe ou de grammaire, ça existe aussi en informatique, et pour un programme, c'est généralement fatal ! Voici quelques exemples :

```
a = 88tt
```

La valeur présente à droite ne correspond pas à un nombre. Elle ne peut être un nom de variable car les noms de variables ont interdiction de commencer par un chiffre. À ce niveau, l'erreur est détectée au bon endroit.

```
a = 88tt1
      ^
SyntaxError: invalid syntax
```

La situation reste identique si nous faisons l'erreur à gauche de l'affectation :

```
77tt = 5
  ^
SyntaxError: invalid syntax
```

2. Variable inconnue

Parfois, cela arrive, nous utilisons une variable jamais définie :

```
print(a)
```

Le message d'erreur est correct et vous indique clairement le problème :

```
print(a)
NameError: name 'a' is not defined
```

Lorsque vous êtes en mode Python interactif, vous conservez vos variables durant toute la session. Ainsi, si dans un exercice précédent vous avez tapé `a = 4`, la variable `a` est toujours définie et vous n'obtiendrez pas le message d'erreur attendu. Pour cela, vous devez quitter puis relancer l'environnement Python.

3. Les erreurs d'indentation

Le langage Python est assez strict sur ce point. Si vous changez votre indentation dans le même branchement, l'erreur est immédiate :

```
if 4 > 3 :
  print("Bonjour")
    print("hello")
```

Voici le message correspondant :

```
print("hello")
^
IndentationError: unexpected indent
```

4. Les erreurs de parenthésage

Le parenthésage (jargon d'informaticien) consiste à associer une parenthèse fermante à toute parenthèse ouvrante. Le cas d'erreur le plus fréquent consiste à oublier la parenthèse fermante en fin de ligne, comme dans l'exemple suivant :

```
a = 5 * (3 + 8
print("toto")
print("titi")
```

Dans ce cas, l'interpréteur Python perd pied et ne comprend pas ce qu'il se passe. Il signalera l'erreur seulement à la ligne suivante lorsqu'il rencontrera la parenthèse fermante recherchée. Il ne comprend alors pas pourquoi une

fonction `print()` se retrouve au beau milieu d'une expression. Voici le message d'erreur associé :

```
print("toto")
   ^
SyntaxError: invalid syntax
```

Votre attitude doit être la suivante : lorsqu'une erreur est signalée sur une ligne semblant tout à fait correcte, il faut chercher l'erreur à la ligne précédente. Parfois, l'erreur peut se situer plusieurs lignes au-dessus.

5. L'oubli du range

Parfois, on oublie la syntaxe de la boucle `for` ou on la mélange avec celle d'un autre langage. Ainsi, on peut écrire :

```
for i in 10 :
  print(i)
```

Dans ce cas précis, vous allez obtenir un message d'erreur qui décoiffe un peu :

```
for i in 10:
TypeError: 'int' object is not iterable
```

L'interpréteur vous signale qu'une valeur de type 'int' n'est pas itérable. Procédons pas à pas. En parlant du type int, l'interpréteur désigne le nombre 10. Le problème est qu'il n'est pas itérable, ce terme sous-entend que 10 ne peut être parcouru. En effet, à cette position, on aurait dû trouver écrit : `range(10)`, soit une plage de valeurs de 0 à 9. Une plage peut être parcourue par une boucle `for` ; par contre, ce n'est pas le cas d'un nombre seul.

6. Les erreurs non signalées

Elles existent et ce sont les plus difficiles à détecter, car elles passent sous les radars, aussi bien pour l'interpréteur Python que pour vos yeux. Il faut bien comprendre que parfois, une faute de frappe aboutit à un code incorrect par rapport à ce que vous vouliez faire, mais valide par rapport à la syntaxe du langage. Ainsi, il n'y a pas de message d'erreur, car il n'y a pas d'erreur sur le plan syntaxique. Mais, étrangement, rien ne se passe comme prévu dans votre programme, il y a un couac quelque part et il est bien caché. Voici un exemple où l'on teste deux âges de 10 et 20 ans pour savoir qui est majeur ou mineur :

```
Age = 10
if Age >= 18 :
  print("Majeur")
else :
  print("Mineur")
Age == 20
if Age >= 18 :
  print("Majeur")
else :
  print("Mineur")
```

Quel est le résultat de ce programme ? Vous pensez à "Mineur" puis "Majeur" ? Cela aurait dû être le cas. Pourtant, vous avez comme affichage : "Mineur" et "Mineur". Peut-être l'avez-vous vu, mais nous aurions dû écrire à la sixième ligne `Age = 20`, et non `Age == 20`. Le symbole == correspond au test d'égalité, l'interpréteur Python effectue donc le test 10 == 20. Le résultat retourné est faux et aucune variable ne le récupère. Cela pourrait sembler étrange, mais c'est toléré et aucun message d'erreur ne ressort. Ainsi, lors du test `Age >= 18`, la variable `Age` est inchangée et vaut toujours 10. Sacré piège, ce symbole == !