Python ≥ 3.11

Le langage de programmation

Introduction

Voici de quoi s'initier à la programmation avec le langage Python.
Actuellement j'ai sous la main la version 3.11, alors…

Principes de base

Python est un langage de programmation. Du texte écrit dans ce langage va permettre d'exprimer la considération d'informations numériques (nombres, textes, etc.) et des traitements de ces informations (calculs, etc.). Avec des librairies supplémentaires, il sera également possible d'interagir avec d'autres logiciels et indirectement avec les différents composants de la machine ou encore avec certains périphériques et l'utilisateur.

Exemple

```
>>> longueur = 5
>>> largeur = 3
>>> superficie = longueur * largeur
>>> print( 'Superficie :', superficie, 'm²' )
Superficie : 15 m²
```

N. B. l'invite `>>>` typique de Python en mode interactif.

Attention que ce n'est pas toujours comme ça. Par exemple, dans Spyder nous avons `In [1]:` pour la première « console ».

Ceci ↑ est la manière interactive d'utiliser Python. Nous écrivons des instructions, Python exécute, interprète, et nous pouvons directement observer d'éventuels résultats textuels. Cela permet de faire des essais ou des calculs directement.

L'autre option est d'écrire dans un fichier *texte.py* et de faire exécuter ensuite ce fichier par Python. Cela permet d'écrire un programme que l'on pourra mettre au point et exécuter plusieurs fois.

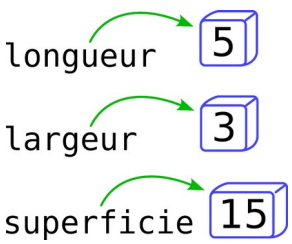
Exemple

```
# superficie.py

longueur = 5
largeur = 3
superficie = longueur * largeur
print( 'Superficie :', superficie, 'm²' )
```

L'univers d'objets

Avec Python, nous évoluons dans un univers d'objets. Dans notre exemple, nous avons créé trois objets de type int.



Mots clé

Voici déjà la liste des mots clé. Ce sont les mots du langage. Leur signification et leurs usages sont déterminés et protégés par le langage.

Attention que la casse (majuscules ou minuscules) est importante. **False**, **True** et **None** représentent des valeurs particulières (vrai, faux et aucun) et commencent par une lettre majuscule.

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

Vous rencontrerez de nombreux autres mots, ou plutôt des identifiants, qui désigneront des objets en fonction des librairies importées et de vos propres écritures interprétées.

Opérateurs

Les opérateurs sont les symboles qui pourront avoir une signification.

= (affectation)	**//** et **//=** (entière)	**&** et **&=**	**.**
+ et **+=**	**%** et **%=** (reste)	**<<** et **<<=**	**<**
- et **-=**	**~**	**>>** et **>>=**	**>**
***** et ***=**	**|** et **|=**	**( )**	**<=**
****** et ****=** (exposant)	**^** et **^=**	**{ }**	**>=**
/ et **/=**		**[ ]**	**==** (égale)
		:	**!=** (différent)

Un opérateur peut être unaire ou binaire en fonction du nombre d'éléments impliqués, un ou deux.

Exemple à ne pas suivre !

Voici trois lignes de code très vilaines, car les noms de variables sont très ambigus et l'on profite de l'**importance de la casse** pour pouvoir faire cela.

```
>>> IF = 10 ; AND = 4 ; OR = 15 ; false = True
>>> if AND < OR and IF < AND or false :
...     print( AND, OR, IF )
...
4 15 10
```

N. B. les deux points **:** qui marquent ensuite la considération d'un sous-ensemble d'instructions, ici en lien avec une condition exprimée après le mot clé **`if`**.

N. B. l'indentation ici constituée de 4 espaces (par défaut en Python) qui permettent de marquer la considération d'un sous-ensemble d'instructions.

`IF`, `AND`, `OR` et `false` peuvent en effet être des noms de variables alors que **`if`**, **`and`**, **`or`** et **`False`** sont des mots clés que l'on ne peut pas redéfinir.

`IF` référence l'entier `10`.
`AND` référence l'entier `4`.
`OR` référence l'entier `15`.
`false` référence le booléen de valeur **`True`**.

Dans le **`if`** qui suit, la condition est donc équivalente à

```
4 < 15 and 10 < 4 or True
```

, évaluée à **`True`** (vrai) car l'opérateur **`or`** est évalué après **`and`**.

Builtins

Au démarrage de Python, toute une série d'objets sont importés en tant qu'objets intégrés. (fonctions et classes)

abs	divmod	isinstance	*property*
all	enumerate	issubclass	*range*
any	eval	iter	repr
ascii	exec	len	reversed
bin	filter	*list*	round
bool	*float*	locals	*set*
breakpoint	format	map	setattr
bytearray	*frozenset*	max	*slice*
bytes	getattr	*memoryview*	sorted
callable	globals	min	*staticmethod*
chr	hasattr	next	*str*
classmethod	hash	*object*	sum
compile	help	oct	super
complex	hex	open	*tuple*
delattr	id	ord	*type*
dict	input	pow	vars
dir	*int*	print	*zip*

Ce sont des fonctions et classes prédéfinies et sur lesquels nous devrions pouvoir compter. Ce ne sont néanmoins pas des mots clés réservés et ils ne sont pas "protégés". Il faudra donc faire attention à ne pas les "écraser" comme dans ce mauvais exemple ↓

```
>>> a = print
>>> type( a )
<class 'builtin_function_or_method'>
>>> print = "Bonjour" # La fonction print a été écrasée:(
>>> type( print )
<class 'str'>
>>> a( print ) # c'est le monde à l'envers :/
Bonjour
>>> ( print, a ) = ( a, print ) # échangeons a et print
>>> print( a ) # Voilà ce qui n'aurait pas dû changer :)
Bonjour
```

Par contre, redéfinir `print` peut être utile malgré tout, dans certaines situations…

Variables, identifiants, types, objets et valeurs

En python, nous travaillons avec des objets. Notre programme lui-même est un objet, chaque fonction définie est un objet. Ce que l'on nomme communément une variable est un objet référencé par un nom.

Un objet est d'un type et possède une valeur. En fonction du type, cette valeur pourra évoluer et l'on dira qu'elle est mutable, ou ne pourra pas évoluer et l'on dira qu'elle est immuable.

Un identifiant référence un objet, ou « rien » (l'objet *None*).

Exemple :

```
>>> a = 5
↓
```

Après cela ↑, l'identifiant *a* référence l'objet de type *int*, immuable, de valeur *5*.

```
↓
>>> type( a )
<class 'int'>
>>> print( a )
5
```

Deux identifiants peuvent référencer le même objet. Mais cela n'a de réelles conséquences que lorsque l'objet est d'un type mutable.

En effet :

```
>>> a = "Bonjour"
>>> b = a
>>> print( b )
Bonjour
↓
```

Les deux identifiants, *a* et *b*, référencent le même objet de type *str*, **immuable**, de valeur "Bonjour".

Immuable signifie que rien ne permet de modifier la valeur de l'objet. Il y a bien des méthodes applicables pour modifier la chaîne de caractères mais ces méthodes ne modifient pas l'objet, elles créent et retournent un nouvel objet avec la nouvelle valeur.

```
↓
>>> a.replace("Bon", "Mauvais ") # le résultat est ici perdu
'Mauvais jour'
>>> print( a )
Bonjour
>>> a = a.replace("Bon", "Mauvais ") # nouvelle affectation pour a
>>> print( a )
Mauvais jour
>>> print( b ) # b référence toujours l'objet str initial
Bonjour
```

Par contre :

```
>>> a = ["Bonjour"] # une liste d'un élément de type str
>>> b = a
>>> a[0] = a[0].replace("Bon","Mauvais ")
>>> print( b[0] )
Mauvais jour
```

Les deux identifiants, *a* et *b*, référencent le même objet de type *list*, **mutable**, de valeur « un élément de type *str* , valeur "Bonjour" ».

Lorsque nous modifions cet élément ([0] désigne le premier élément de la liste), en lui affectant le résultat de la méthode replace, un nouvel objet de type *str*, nous modifions bien l'objet de type *list*, le même objet de type *list* toujours référencé par *a* et par *b*.

Types intégrés (built-in)

Entier *int*

```
>>> a = 5
>>> type( a )
<class 'int'>
>>> a = 3546879442324561021588971316549872316576543214687
>>> a
3546879442324561021588971316549872316576543214687
>>> type( a )
<class 'int'>
```

En Python 3, le type entier (*int*) n'a pas de limite. La grandeur d'un nombre entier, nombre de chiffres, n'a aucune limite, mises à part les performances et la mémoire disponible.

Réel *float*

En programmation d'origine anglophone, le point remplace notre virgule pour séparer la partie décimale de la partie entière.

```
>>> a = 5.
>>> type( a )
<class 'float'>
>>> print( a )
5.0
>>> a = .5
>>> print( a )
0.5
>>> a = 3.14159265358979323846
>>> a
3.141592653589793
```

N. B. que nous ne sommes pas obligé d'écrire l'éventuel zéro (« non significatif ») avant ou après le point.

N. B. que nous ne retrouvons donc pas toutes les décimales renseignées lors de l'affectation.
(16 chiffres significatifs)

Le type réel (float) est soumis à une limite dans le nombre de chiffres significatifs (16) et également une limite quant « au nombre de zéros avant ou après » (± 300 zéros)…

Un float peut également avoir les valeurs *inf*, *-inf* et *nan* pour respectivement infini, moins l'infini et "pas un nombre".

```
>>> a = 1e300 # ← écriture scientifique
>>> b = 1.*10**300 # ← 10 exposant 300
>>> a == b
True
>>> print( a, b )
1e+300 1e+300
>>> a *= 1_000_000_000
>>> a
inf
>>> a = float("nan") # conversion str → float
>>> a
nan
```

N. B. les blancs soulignés ("underscores") dans le nombre, ici en guise de séparateur de milliers.

Complexe *complex*

```
>>> a = (-1)**(1/2) # Racine carrée de -1
>>> type( a )
<class 'complex'>
>>> a = 0+1j
>>> a
1j
>>> a **= 2
>>> a
(-1+0j)
>>> a.real
-1.0
>>> a.imag
0.0
```

Le type complexe fait référence aux nombres complexes créés comme extension de l'ensemble des nombres réels, contenant en particulier un nombre imaginaire noté ici *j* tel que $j^2 = -1$.

.real et *.imag* donne respectivement accès à la partie réelle et la partie imaginaire en tant que *float*.

Chaîne de caractères *str*

```
>>> a = "Bonjour"
>>> type( a )
<class 'str'>
>>> b = " :)"
>>> c = a + b
>>> c
Bonjour :)
```

Les chaînes de caractères sont encodés en UTF-8.

Octets *bytes*

Ce type permet de manipuler des octets (“bytes” en anglais). Voici par exemple des conversions avec des chaînes de caractère (UTF-8)

```
>>> a = b"Bonjour"
>>> type( a )
<class 'bytes'>
>>> b = " :)"
>>> c = a + b
>>> c
Bonjour :)
```

Booléen *bool*

En programmation, nous aurons souvent besoin de distinguer le vrai du faux. Cela constitue un type en soi. Vous l'aurez compris, un booléen ne peut avoir qu'une des valeurs True et False, respectivement vrai ou faux.

```
>>> a = 5 < 0 # 5 est-il plus petit que 0 ?
>>> type( a )
<class 'bool'>
>>> a
False
>>> a = (not 5 < 0) & (not 8 > 15)
>>> a # 5 n'est pas < que 0 et 8 n'est pas > que 15
True
>>> a = "Bonjour"
>>> b = "Bonjour"
>>> a is b # est-ce le même objet ?
```

Tuple *tuple*

Un tuple est une combinaison de plusieurs objets. Un tuple est **immuable** et peu servir à manipuler plusieurs objets en même temps.

```
>>> a = (5, 8)
>>> type(a)
<class 'tuple'>
>>> a
(5, 8)
>>> (b, c) = (9, 1) # affectation parallèle
>>> (b, c) = (c, b) # échange entre b et c
>>> (b, c)
(1, 9)
>>> a = tuple() # pour un tuple vide
>>> a
()
>>> tutu = (1,2,3,4,5,6)
>>> taille_tutu = len( tutu )
>>> taille_tutu
6
```

Une liste tuple peut représenter les **arguments non-nommés** passés à une fonction.

```
>>> arguments = ("Lundi", 5, "septembre")
>>> print( *arguments )
Lundi 5 septembre
```

Nous verrons plus loin comment récupérer les arguments d'une fonction sous cette forme.

Les **paracentèses** ne sont **pas toujours obligatoires**. Néanmoins je vous conseille de les garder lorsque cela permet de faciliter la bonne compréhension du code.

Liste *list*

Une liste est **mutable** et peut donc évoluer.

```
>>> a = [5, 8, 5]
>>> type(a)
<class 'list'>
>>> a
[5, 8, 5]
>>> a.append( 9 ) # ajouter un élément à la fin
>>> a
[5, 8, 5, 9]
>>> len( a )
4
>>> a.pop() # retirer le dernier élément
9
>>> a
[5, 8, 5]
>>> a.pop(1) # retirer l'élément d'indice 1
8
>>> a
[5, 5]
```

Une liste vide peut être défini ainsi,

```
>>> a = []
>>> b = list()
>>> type(a) ; type(b)
<class 'list'>
<class 'list'>
```

Un tuple peut-être construit à partir d'une liste et inversement. Cela peut être obtenu implicitement lors du page d'arguments...

```
>>> a = (5, 9, "Bonjour")
>>> b = list( a )
>>> b
[5, 9, 'Bonjour']
>>> b.append(8)
>>> c = tuple( b )
>>> c
(5, 9, 'Bonjour', 8)
>>> arguments = ["Lundi", 5, "septembre"]
>>> print( *arguments )
Lundi 5 septembre
```

Ensemble *set*

Un ensemble (*set*) contient des objets uniques et ordonnés de manière arbitraire. Il est mutable et peut donc évoluer.

```
>>> a = {1, 4, 3, 4, 2}
>>> type(a)
<class 'set'>
>>> a
{1, 2, 3, 4}
>>> a.add( 9 ) # ajouter un élément
>>> len( a )
5
>>> a
{1, 2, 3, 4, 9}
>>> a.pop() # retirer un élément arbitrairement
1
>>> a
{2, 3, 4, 9}
```

Dictionnaire *dict*

Un dictionnaire est un ensemble d'association entre une clé ("key") et une valeur ("value").

```
>>> a = {'oui':'yes', 'non':'no', 'salut':'hi'}
>>> type(a)
<class 'dict'>
>>> a
{'oui': 'yes', 'non': 'no', 'salut': 'hi'}
```

Un dictionnaire peut-être être utilisé pour passer des arguments nommés à une fonction.

```
>>> def mafonction( a, b, c ) :
...     print( a, b, c )
>>> a = {'b':'toi', 'c':':)', 'a':'Bonjour' }
>>> mafonction( **a )
Bonjour toi :)
```

Indentation

L'indentation en python à un rôle primordial structurel. Nous sommes obligés d'indenter et nous ne pouvons pas indenter n'importe où et n'importe comment.

L'indentation est probablement paramétrable et il est probablement possible d'utiliser le caractère tabulation. Mais l'usage le plus courant et configuré par défaut consiste à utiliser **4 espaces**.

Nous allons voir ci-dessous comment interviennent les indentations. Vous constaterez par ailleurs que de nombreux environnements de travail proposent une indentation automatique. Il est donc rare, en pratique, de devoir appuyer soi-même quatre fois sur la barre d'espace.

Fonctions

Du code python peut-être exécuter directement, soit de manière interactive, soit dans un fichier, « du haut vers le bas ». Mais il est également possible de définir des fonctions d'une part et de les appeler ensuite, une fois, deux fois, etc. Une fonction peut en appeler une autre, etc.

Exemple :

```
>>> def mafonction() :
...     print("Coucou depuis mafonction:)")
...
>>> mafonction()
Coucou depuis mafonction:)
>>> type( mafonction )
<class 'function'>
```

if elif else

Si nous devions ne pas toujours faire la même chose, mais tenir compte de quelques conditions pour faire ceci ou cela, nous utiliserions les mots clé *if*, *elif* et *else*.

Attention, je vous propose ici de travailler avec un fichier à exécuter en une fois, et non plus le mode interactif qu'indiquent sinon les trois symboles >>>.

```
import random # aléatoire

a = random.random()

if a < .33 :
    print("Il est petit ce a =", a) # héhé
elif a < .66 :
    print("Il est moyen ce a =", a)
else :
    print("Il est grand ce a =", a)
```

Exécutez ce petit bout de code plusieurs fois et obtenez l'un des trois types résultats suivant :

```
Il est petit ce a = 0.14042961846617863
```

ou

```
Il est moyen ce a = 0.5528754741160663
```

ou

```
Il est grand ce a = 0.964085495963503
```

if et **else** peuvent également servir à déterminer un objet à considérer.

```
>>> age = 18
>>> print( "Il est", "mineur" if age < 18 else "majeur" )
Il est majeur
```

Boucle `for`

Les boucles sont très importantes dès lors que nous avons de répétition.

La boucle for (« pour ») est adaptée pour parcourir un ensemble d'éléments.

Exemple

```
>>> for i in range(4) :
...     print( "Bonjour", i )
Bonjour 0
Bonjour 1
Bonjour 2
Bonjour 3
```

Parcourir les éléments d'un dictionnaire ↓

```
>>> a = {'oui':'yes', 'non':'no', 'salut':'hi'}
>>> for k, v in a.items() :
...     print( k, v )
oui yes
non no
salut hi
```

Boucle `while`

La boucle while (« tant que ») est adaptée lorsqu'il faut faire et refaire tant qu'une condition est "True".

Exemple

```
>>> i = 4
>>> while 0 < i :
...     print("Bonjour", i)
...     i -= 1
...
Bonjour 1
Bonjour 2
Bonjour 3
Bonjour 4
```

break permet de sortir d'une boucle, tandis que continue permet de passer à l'élément suivant ou de revenir à l'évaluation de la condition (boucle while).

Exemple

```
>>> i = 0
>>> while True :
...     print("Bonjour", i)
...     i += 1
...     if i % 2 : continue
...     print("impaire !")
...     if 3 < i : break
...
Bonjour 0
Bonjour 1
impaire !
Bonjour 2
Bonjour 3
impaire !
```

Modules

Tout le programme ne doit pas être écrit en un seul fichier et il existe même des choses déjà existantes que l'on peut réutiliser… Ce sont les modules.

Si vous avez deux fichiers Python (.py) dans un même dossier, l'un peu utiliser l'autre en tant que module.

Python propose également tout un ensemble de modules pour pouvoir faire toutes sortes de choses.

Un module se trouve en définitive être un ensemble d'objets Python au sein d'un espace de nom.

```
>>> import math
>>> type( math )
<class 'module'>
>>> math.cos( math.pi / 3 )
0.5000000000000001
```

Il est possible de renommer l'espace de nom obtenu...

```
>>> import math as M
>>> type( M )
<class 'module'>
>>> M.cos( M.pi / 3 )
0.5000000000000001
```

Il est possible d'importer dans l'espace global certains objets provenant d'un module…

```
>>> from math import cos, pi
>>> cos( pi / 3 )
0.5000000000000001
```

Exercices

Voici quelques petits exercices. Des solutions vous sont proposées dans le chapitre suivant.

1) Ce nombre est-il premier ?

Écrivons une fonction qui répond à la question de savoir si un nombre est premier. Le nombre y sera représenté par l'identifiant n qui devrait être un nombre entier.

```
def est_premier( n ) :
    .
    .
    .
```

Si n est un nombre premier, alors la réponse est « oui » et donc ici la valeur retournée **`True`**.

Si n n'est pas un nombre premier, alors la réponse est « non » et la valeur retournée est **`False`**.

1. Décrivez l'algorithme avec un diagramme logique.
2. Rédigez la fonction en python.
3. Est-il possible d'optimiser la fonction ?

2) Le x^{ème} nombre premier

Écrivons une fonction qui nous donnera le x^{ème} nombre premier. La fonction reçoit un argument x qui devra être un nombre premier strictement positif.

```
def xim_premier( x ) :
    .
    .
    .
```

Si x est un entier plus grand que 0, alors la fonction retourne le x^{ème} nombre premier.

Dans un premier temps, convenons que si x est plus petit que 1, alors nous retournons 0. Si x = 1 nous retournons 2 ; x = 2 nous retournons 3 ; etc.

1. Décrivez l'algorithme avec un diagramme logique.
2. Rédigez la fonction en python.
3. Est-il possible d'optimiser la fonction ?

Solutions

Voici les solutions aux exercices proposés au chapitre précédent.

1) Ce nombre est-il premier ?

Une première approche pourrait être ceci, partant de d = 2 vers la racine carrée de n.

```
def est_premier( n ) :
    if n < 2 : return False
    d = 2
    while d <= n**.5 :
        if n % d == 0 : return False
        d + =1
    return True
```

Statistiquement, nous observerons de meilleures performances en partant de la racine carrée de n.

```
def est_premier( n ) :
    if n < 2 : return False
    d = int( n ** .5 )
    while n % d : d -= 1
    return d == 1
```

Pour mesurer et comparer les performances ↓

```
import time
t1 = time.time_ns()
for n in range(1000) : est_premier(n)
t2 = time.time_ns()
print("pour n de 0 à 1000 :", (t2-t1)//1000, "µs" )
```

2) Le x^{ème} nombre premier

Une première approche pourrait donner ceci ↓

```
def xim_premier( x ) :
    if x < 2 : return 2
    n = 3 ; x -= 1
    while x > 1 :
        n += 2
        if est_premier( n ) : x -= 1
    return n
```

Annexes

Mots clés

and est un opérateur logique binaire qui permet d'obtenir **True** à partir de deux **True**, sinon **False**.

as permet de modifier le nom d'un module ou d'un objet importé. Voir également **import** et **from**. **as** est également utilisé avec le mot clé **with** pour donner un nom à la ressource instanciée.

assert permet de rajouter des contrôles pour le débogage d'un programme.

async et **await**
sont impliqué dans les coroutines et des exécutions asynchrones.
(depuis 3.5 et officiellement 3.7)

break sortir d'une boucle (**while** ou **for**)

class définir une classe

continue dans une boucle (**while** ou **for**), interrompre le traitement de l'itération en cours, revenir au point conditionel d'une boucle **while** ou passer à l'éventuel élément suivant d'une boucle **for**.

def définir une fonction.

del détacher un identifiant ("name") d'un éventuel objet et supprimer l'identifiant.

elif après un **if** ou après un **elif**, considérer une condition alternative.

else après un **if** ou après un **elif**, considérer tout autre cas de figure.

except traiter les exceptions attendues.

False la valeur booléenne faux.

finally finaliser un bloque d'essai (**try**)

for boucle sur un ensemble d'éléments

from à partir d'un module, importation de quelques objets...

global définir ou utiliser un identifiant global

if considérer une condition

import importer un module ou quelques objets (après **from**)

in vérifier si un élément se trouve dans un contenant

is vérifier l'unicité d'un objet à partir de deux identifiants (au delà de l'égalité), « Est-ce le même objet ? ».

lambda une fonction sans identifiant, dite « *Lambda* »

None objet de type NoneType.

nonlocal utiliser un identifiant extérieur à une définition de fonction dans une fonction, à l'instar de **global** pour un identifiant global.

not inversion logique.

or opérateur *ou* logique.

pass ne rien faire.

raise lever une exception.

return sortir d'une fonction en retournant un objet.

True valeur booléenne *vrai*.

try	essayer quelques instructions pour lesquels nous pourrons gérer d'éventuelles exceptions (cf. **except** et **finally**)
while	boucle tant qu'une condition est vérifiée (« → vrai »)
with	utiliser de manière sécurisée une *ressource*. Une *ressource* étant instanciée à partir d'une *classe* particulière avec des méthodes d'entrée et de sortie. Ces méthodes seront respectivement appelées au début et à la fin du bloque d'instructions, qu'il y ai ou non une éventuelle exception.
yield	retourne une valeur sans réellement sortir de la fonction. Pour l'appelant, la fonction aura retourné un objet de type *generator*.

Histoire de Python

Guido van Rossum a commencé l'écriture de Python en décembre 1989.

- Python 3.13 : octobre 2024
- Python 3.12 : octobre 2023
- Python 3.11 : octobre 2022
- Python 3.10 : octobre 2021
- Python 3.9 : octobre 2020
- Python 3.8 : octobre 2019
- Python 3.7 : juin 2018
- Python 3.6 : décembre 2016
- Python 3.5 : septembre 2015
- Python 3.4 : mars 2014
- Python 3.3 : septembre 2012
- Python 3.2 : février 2011
- Python 2.7 : juillet 2010
- Python 3.1 : juin 2009
- Python 3.0 : décembre 2008
- Python 2.6 : octobre 2008
- Python 2.5 : septembre 2006
- Python 2.4 : mars 2005
- Python 2.3 : juillet 2003
- Python 2.2 : décembre 2001
- Python 2.1 : avril 2001
- Python 2.0 : octobre 2000
- Python 1.5 : avril 1999
- Python 0.9.1 : 20 février 1991 (date communément utilisée comme date de naissance du projet Python)

Évolutions du langage :

- Python 3.13 : ajoute un compilateur JIT qui pourrait améliorer les performances du langage d'environ 9 %
- Python 3.12 :
  - f-string + guillemets sans échappement et commentaires (PEP 701)
  - compréhensions plus rapides
- Python 3.11 :
  - entre 10 et 60 % plus rapide.
  - Nouvelles fonctionnalités de typage (PEP 646, 655, 673, 675, 681)
- Python 3.10 : Filtrage par motifs structurels (PEP 634, 635 et 636)
- Python 3.9 : opérateur fusion pour dict, ab = a | b ou a |= b. (PEP 584)
- Python 3.8 : x := 1 assignment expression (PEP 572) and / in function for positional-only parameters (PEP 570)
- Python 3.7 : async and await deviennent des mots clés
- Python 3.6 :
  - f-strings (PEP 498 “Literal String Interpolation”)
  - Blancs soulignés dans les nombres (PEP 515 – Underscores in Numeric Literals) 1000000 == 1_000_000 → True
- Python 3.5 :
  - Ajout de async et await (mais qui ne sont pas de vrai mots clés)
  - L’opérateur @ (PEP 465 “A dedicated infix operator for matrix multiplication”)
  - PEP 448 “Additional Unpacking Generalization”
- Python 3.4 : pas de changement
- Python 3.3 :
  - yield from: PEP 380 “Syntax for Delegating to a Subgenerator”
  - u'unicode' syntax is back: PEP 414 “Explicit Unicode literals”
- Python 3.2 : pas de changement
- Python 2.7 : tous les changements de Python 3.1
- Python 3.1 :
  - dict/set comprehension
  - set literals
  - multiple context managers in a single with statement
- Python 3.0:
  - all changes of Python 2.6
  - nouveau mot clé, nonlocal
  - raise exc from exc2: PEP 3134 “Exception Chaining and Embedded Tracebacks”
  - print and exec deviennent des fonctions
  - True, False, None, as, with sont des mots réservés
  - Change from except exc, var to except exc as var: PEP 3110 “Catching Exceptions in Python 3000”

    - Removed syntax: a <> b, `a`, 123l, 123L, u'unicode', U'unicode' and def func(a, (b, c)): pass
    - PEP 3132 “Extended Iterable Unpacking”
- Python 2.6:
    - with: PEP 343 “The “with” Statement”
    - b'bytes' syntax: PEP 3112 “Bytes literals in Python 3000”