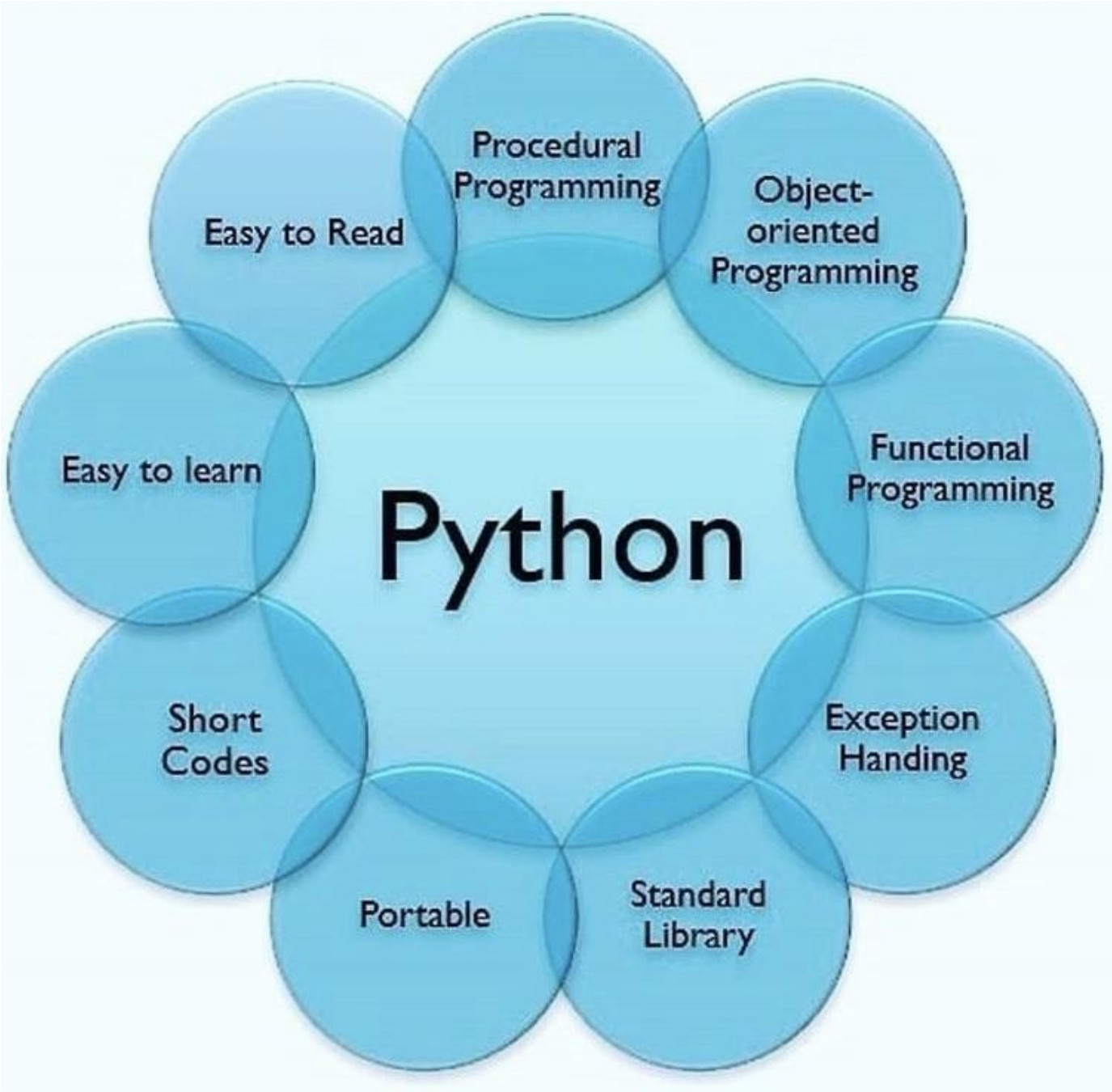
Un langage interprété pour usages multiples

Objectifs

- Faire connaissance avec Python
- Écrire des scripts simples ou avancés



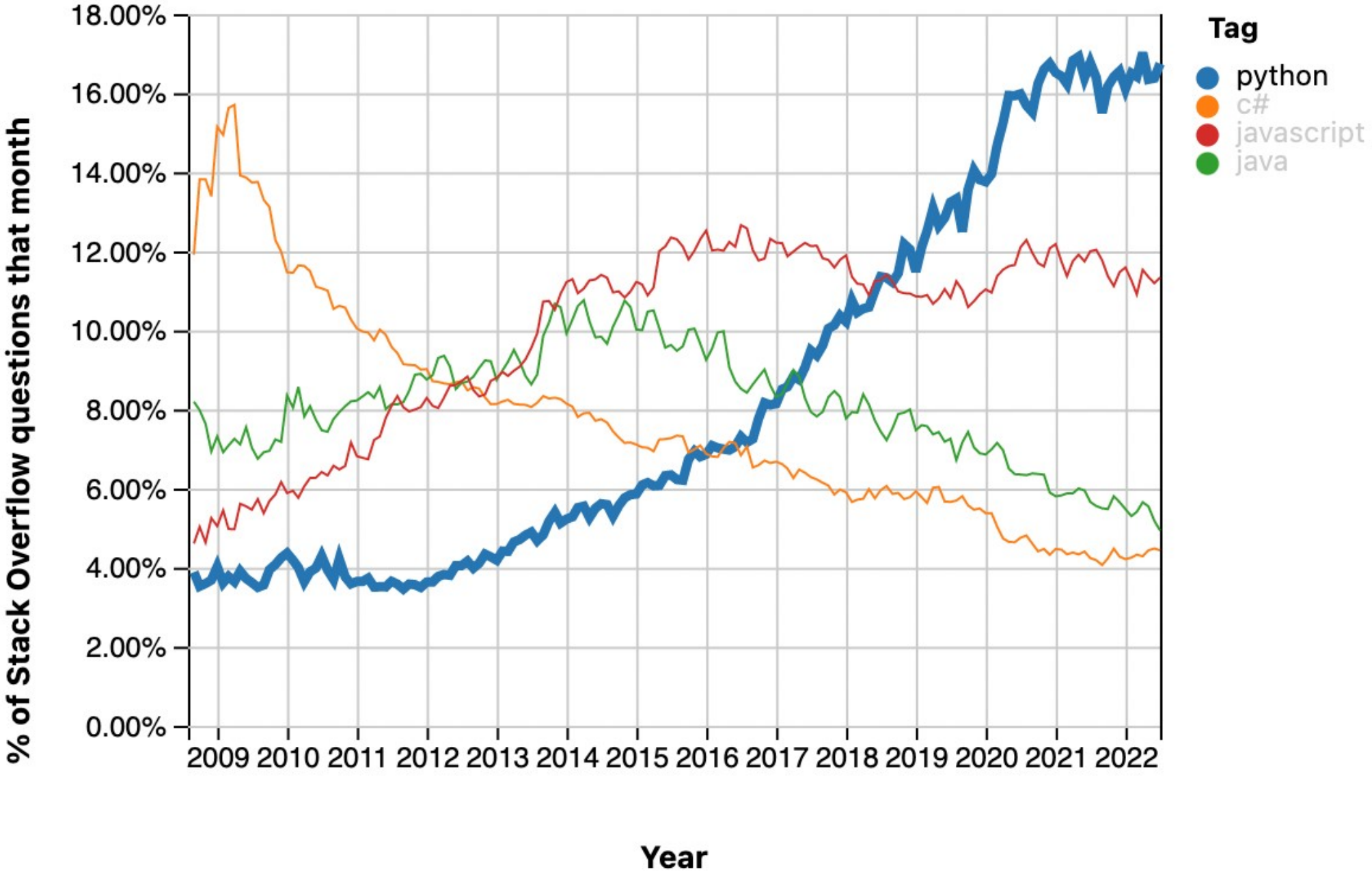
Python

- Créé par Guido van Rossum en 1989
- Langage de haut niveau
- Langage à typage dynamique
- Langage semi-interprété
- Langage procédural (impératif)
- Langage orienté objet

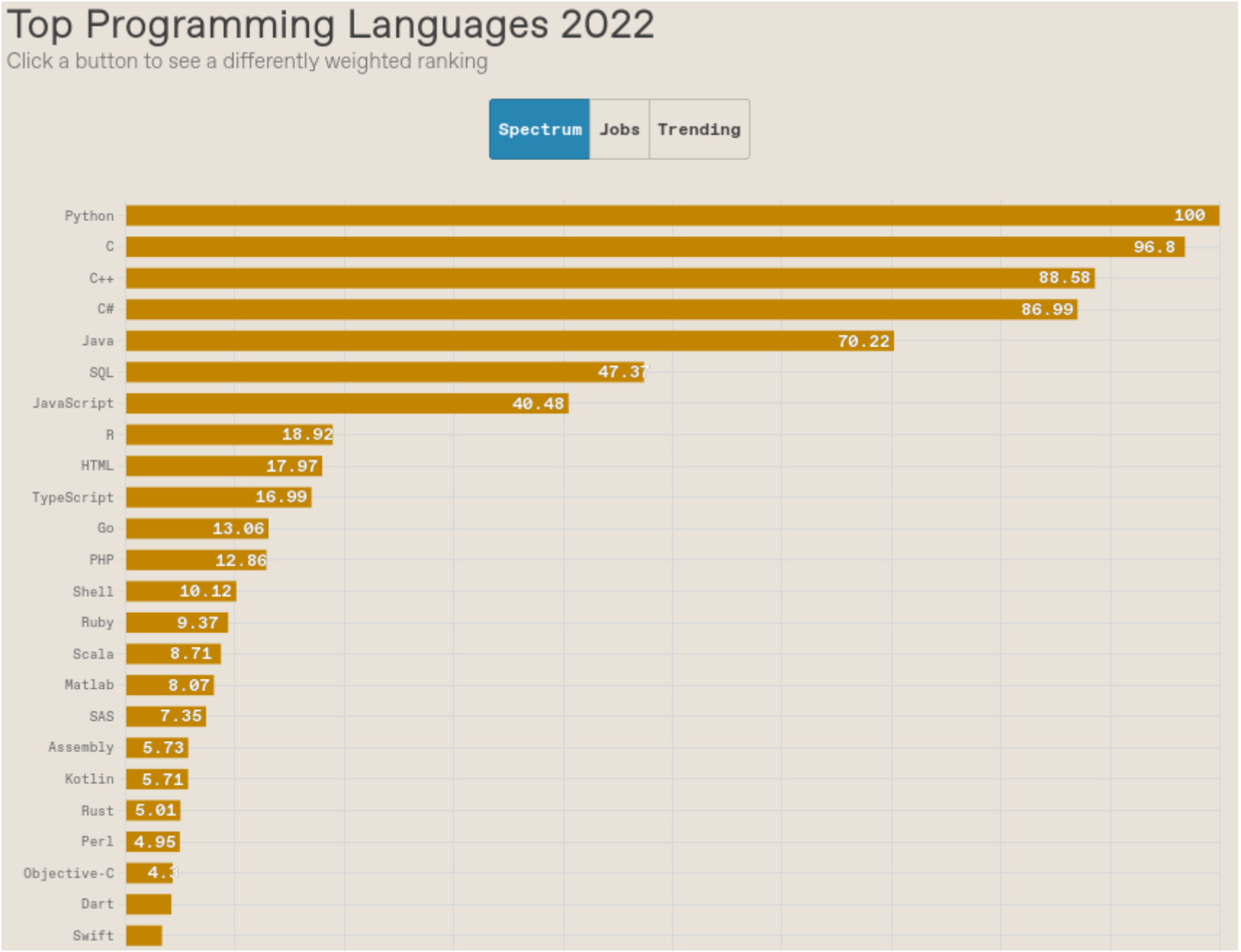
Python et les offres d'emploi

(1) Java – 65,986 jobs
(2) **Python – 61,818 jobs**
(3) Javascript – 38,018 jobs
(4) C++ – 36,798 jobs
(5) C# – 27,521 jobs
(6) PHP – 16,890 jobs
(7) PERL – 13, 727 jobs

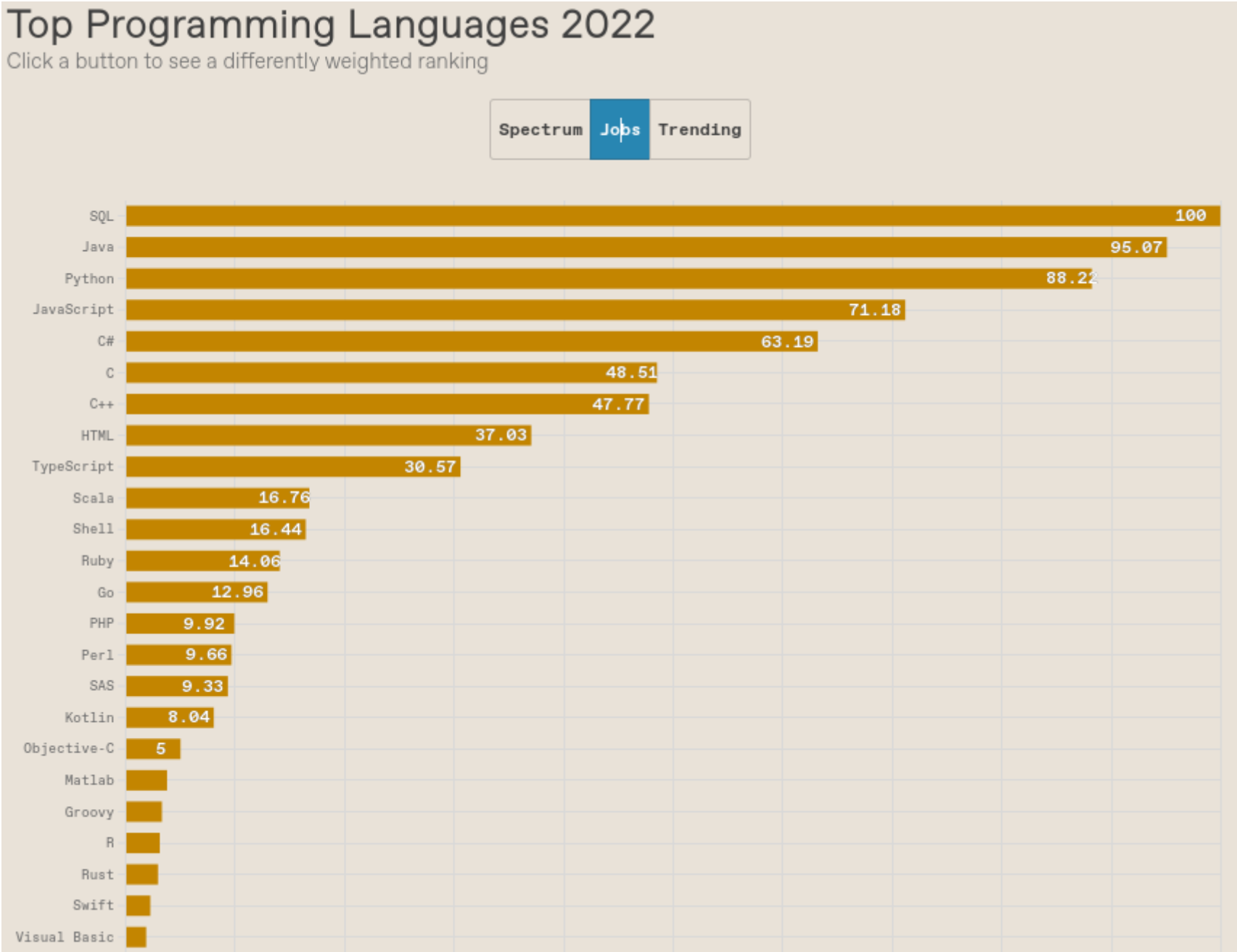
Source : codingdojo.com – Mars 2019



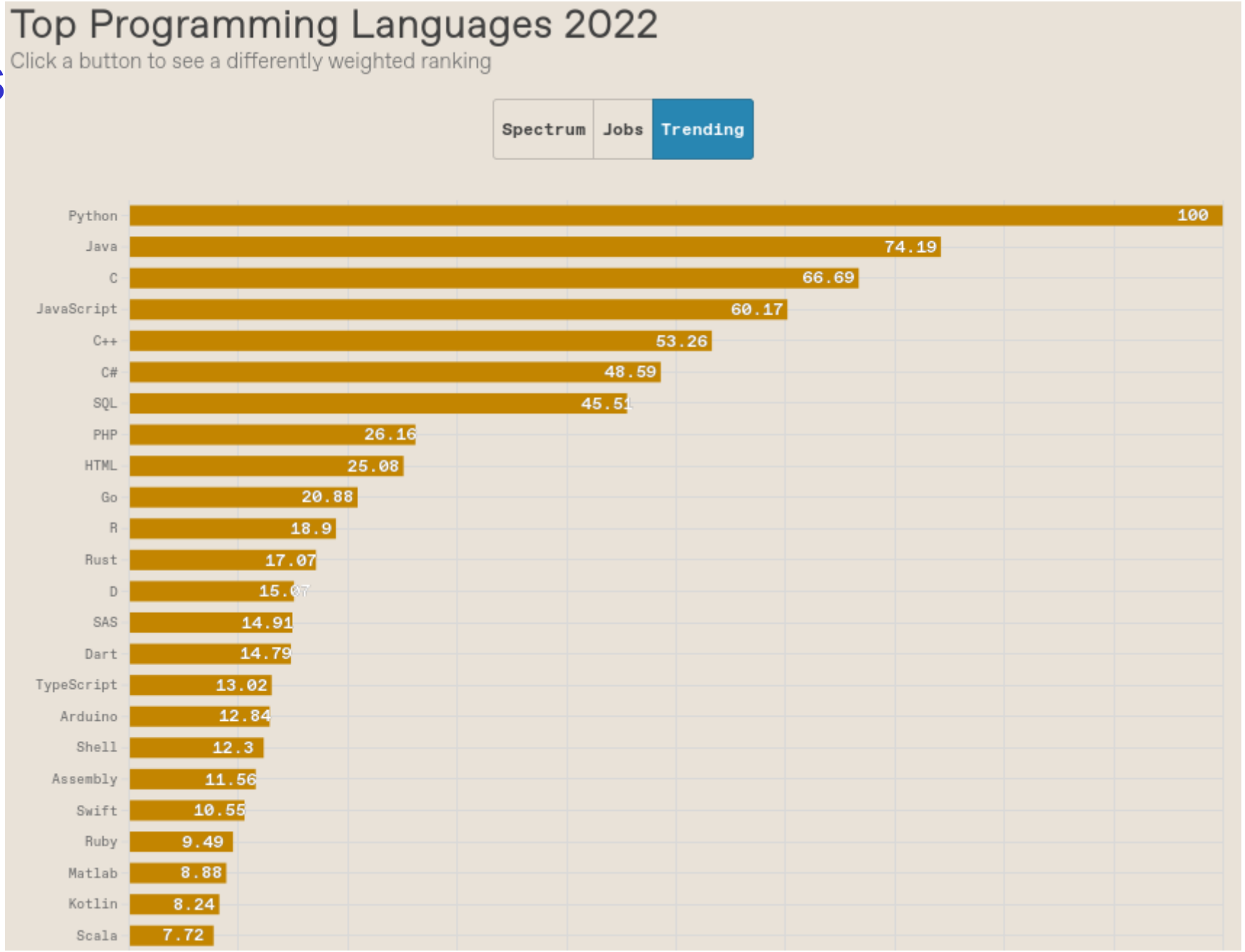
Spectre selon IEEE



Jobs selon IEEE



Tendances selon IEEE



Avantages / inconvénients

- Productivité (concentration sur l'objectif et non sur le code)
- Diversité : 10.000+ modules
- Multiplateforme
- Open Source

- Vitesse < langage compilé (C)
- Pour des adultes consentants _

Python pour quels usages ?

- Applications
- Services web
- Scripts systèmes
- Outils systèmes et réseaux
- Jeux

Évolutions de Python

- La Python Software Foundation (PSF) est en charge du développement du langage
- Les Python Enhancement Proposal (PEP) proposent des :
  - Améliorations techniques
  - Recommandations
  - Exemple : Normes de style codage : PEP0008 https://www.python.org/dev/peps/pep-0008/
- https://linformaticien.com/ameliorer-son-code-python/

Versions de Python

- 2.x : version très répandue, mais la transition vers 3.x est en cours. L'arrêt du support de la version 2 est le 1/1/2020 !
- 3.x : version actuelle qui corrige ses défauts de jeunesse. Non rétro-compatible avec 2.x

Nous ferons le choix de 3.10 ou postérieur

Qu'est-ce qu'un script ?

- Un programme de haut niveau souvent court écrit dans un langage de script :
  - Shell
  - Perl
  - Ruby
  - Python
  - Javascript
  - ...

Caractéristiques d'un script

- 'Glue' entre des programmes
- Traitement intensif de chaînes de caractères
- Manipulation de fichiers et répertoires
- Un code adapté à un problème
- Quelques scripts permettent de maîtriser un gros systèmes
- Interface graphique optionnelle
- Portable sur divers OS
- Pas de compilation

Python

Les bases

Programme Hello world

```
#!/usr/bin/env python


print "Hello, World! Python 2.x"
ou
print ("Hello, World!Python 3.x")
```

Indentation : fondamentale

- L'indentation fait partie du langage, on associe la notion de bloc et présentation
- Pas de `{}` pour marquer les blocs mais
  - caractère `:` en fin de ligne
  - indentation

Les types numériques

- Entier, nombre sur 32 bits
- Entier long, nombre supérieur à 32 bits
- Flottant : nombre décimal
- Complexe : nombre avec partie réelle et imaginaire

Les chaînes de caractères

- Les chaînes utilisent les simple ou double quotes

```
A = "Salut double"

B = 'Salut simple'
```

- Concaténation

```
C = A + B
```

- Répétition

```
C = 3 * A
```

Les chaînes brutes (raw strings)

- Les chaînes peuvent être brutes *i.e* sans interprétation des métacaractères
- On préfixe la chaine par le caractère modificateur `r` (*raw*)
- Évite l'échappement des métacaractères

```
REP = "C:\\\\windows"

REP = r"C:\Windows"
```

Les chaînes sont immuables (*immutable)*

- Lorsque une valeur est affectée à une variable de type chaine, celle-ci ne peut être changée élément par élément
- Exemple :

```
X="Jaste pour le test"

X[1]='u'

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

X="Juste pour le test"
```

Opérateur et fonctions

Opération	Syntaxe	Fonction
Addition	`a + b`	`add(a, b)`
Concaténation	`seq1 + seq2`	`concat(seq1, seq2)`
Test d'inclusion	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Et bit à bit	`a & b`	`and_(a, b)`
Ou exclusif bit à bit	`a ^ b`	`xor(a, b)`
Inversion bit à bit	`~ a`	`invert(a)`
Ou bit à bit	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identité	`a is b`	`is_(a, b)`
Identité	`a is not b`	`is_not(a, b)`
Affectation par index	`obj[k] = v`	`setitem(obj, k, v)`
Suppression par index	`del obj[k]`	`delitem(obj, k)`
Indexation	`obj[k]`	`getitem(obj, k)`
Décalage bit à bit gauche	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Multiplication matricielle	`a @ b`	`matmul(a, b)`
Opposé	`- a`	`neg(a)`
Négation (logique)	`not a`	`not_(a)`
Valeur positive	`+ a`	`pos(a)`
Décalage bit à bit droite	`a >> b`	`rshift(a, b)`
Affectation par tranche	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Suppression par tranche	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Tranche	`seq[i:j]`	`getitem(seq, slice(i, j))`
Formatage de chaînes de caractères	`s % obj`	`mod(s, obj)`
Soustraction	`a - b`	`sub(a, b)`
Test de véracité	`obj`	`truth(obj)`
Ordre	`a < b`	`lt(a, b)`
Ordre	`a <= b`	`le(a, b)`
Égalité	`a == b`	`eq(a, b)`
Inégalité	`a != b`	`ne(a, b)`
Ordre	`a >= b`	`ge(a, b)`
Ordre	`a > b`	`gt(a, b)`

P. MALINGE

Les structures de contrôle : `if`

```
if answer == "copy":
   copyfile = "tmp.copy"
elif answer == "run" or answer == "execute":
   run = True
elif answer == "quit" and not eps < eps_crit:
   quit = True
else:
   print ("Invalid answer%s" % answer)
```

Les structures de contrôle : `for`

```
for x in range(0, 10):
  print (x)
```

Les structures de contrôle : while, break

```
f = open(filename, 'r')
while 1:
  line = f.readline()
  if line == '':
    break
  # process line
...
```

Les structures de contrôle : `for`

```
files = os.listdir(os.curdir) # Liste le contenu
for file in files:
  if not os.path.isfile(file):
    continue # not a regular file, continue
         with next
```

Les fonctions

- On définit une fonction de la manière suivante :
  - Le mot clé `def`, suivi du nom de la fonction
  - Les paramètres éventuels
  - Les caractères `->`
  - Le type de l'objet retourné
  - Le caractère :

```
def f(x) -> int :
  return x**2
```

Le retour de fonction

- `->` permet de connaître le type de ce qui est retourné
- Remarque : une fonction qui n'a pas de return, renvoie malgré tout `None`

Des paramètres par défaut ?

- Les valeurs de certains paramètres peuvent être fixées si les valeurs ne sont pas fournies lors de l'appel.
- Ce doit être les derniers paramètres

```
def g(nom, prénom, sexe='M') :
    ...
```

Ordre des paramètres

- L'ordre des paramètres n'est pas forcément fixé si l'on peut les nommer lors de l'appel

```
g(sexe='F', prenom='Bob', nom='Bi'):
    ...
```

Retour(s) de fonction

- Une fonction retourne une ou plusieurs valeurs

```
def g(nom, prenom, sexe='M') :
    ...
    return nom, prenom
```

Dans l'appel de fonction, on récupère les retours :

```
ntempo, ptempo = g('Black', 'Joe', 'M')
```

Passage par valeur ou adresse

- En Python tous les paramètres sont passés par **adresse** !
- Attention cependant aux types non modifiables, on aura l'impression que le passage a été par valeur !

Les modules

- On peut regrouper les fonctions en modules
- Le mot clé `import` permet de charger les fonctions
- 3 manières d'importer :
  - `from math import *` : on importe tout
  - `from math import sin, cos` : on importe seulement `sin` et `cos`, les autres sont inaccessibles
  - `import math` : on importe tout mais on devra préfixer l'appel de la fonction par le nom du module

Les types complexes

- Les listes
- Les ensembles
- Les tuples
- Les dictionnaires

Les listes

- Séquence d'éléments, modifiable et ordonnée.
- Les éléments sont des objets quelconques, ils peuvent être de type différents
- Les éléments de la liste sont décrits à l'intérieur de crochets et sont séparés par une virgule
- Ex. : `x = ['1', 'blabla', 67]`

Les listes (suite)

- On peut créer une liste :

`x = list('but')` ce qui crée la liste : `['b','u','t']`

- Une liste vide est crée par `[]`
- Les éléments d'une liste peuvent être accéder par leur indice dans la liste :
- `x[0]` : premier élément
- `x[-1]` : dernier élément
- `x[-2]` : avant dernier

Les listes (suite)

- `X[n:p]` : les éléments de *n* inclus à *p* exclu
- `X[:n]` : les *n* premiers éléments
- `X[n:]` : les *n* derniers éléments
- Exemples :

```
0123456789...

X=list('il fait beau')

X[3:6]  ⇒  ['f', 'a', 'i']

X[:6]  ⇒  ['i', 'l', ' ', 'f', 'a', 'i']

X[6:]  ⇒  ['t', ' ', 'b', 'e', 'a', 'u']
```

Copie de listes

- `X=Y` : ne copie pas la liste Y dans X mais X et Y partagent les mêmes données (pointent sur la même liste)
- `X=copy.deepcopy(Y)`: copie les éléments de la liste Y dans X

Les listes en intention

- Bien souvent on vient appliquer des traitements aux éléments d'une liste, on parcourt à nouveau la liste.
- Ex. mettre en majuscule les éléments

```
sequence = ["a", "b", "c"]
new_sequence = []
for element in sequence:
    new_sequence.append(element.upper())
print(new_sequence)
```

Les listes en intention (suite)

- devient

```
sequence = ["a", "b", "c"]
sequence = [element.upper() for element in sequence]
print(sequence)
```

- On crée directement la liste en itérant sur les éléments (ici en plus avec réutilisation de la même liste `sequence`)

Les ensembles

- Ensembles au sens mathématique : une collection d'éléments uniques
- Il en existe 2 types :
  - `Set` : ensemble standard
  - `Frozenset` : (ensemble gelé, non modifiable)

Les tuples

- Ensemble ordonné et immuable d'éléments
- Les éléments du tuple sont décrits à l'intérieur de parenthèses et sont séparés par une virgule
- Ex. : `x = ('1', 'blabla', 67)`
- Un tuple à 1 seul élément : `x = ('Linux',)`
- Quand utiliser un tuple ? Définir des constantes
- Avantage par rapport aux listes : rapidité
- Possibilité de convertir un tuple en liste et réciproquement, exemple :

```
y=list(x)
z=tuple(y)
```

Les dictionnaires (tableaux associatifs)

- Liste non ordonnée dans laquelle l'accès aux éléments se fait par une clé (alphanumérique ou numérique)
- C'est une association `clé:valeur`. La clé doit être unique
- Les éléments du dictionnaire sont décrits à l'intérieur d'accolades et sont séparés par une virgule
- Ex. : `msg = {'ERR_IO':'Erreur I/O', 'ERR_ALLOC':'Erreur allocation'}`

Opérations sur les types complexes

- Liste : ajouter, supprimer, trier, renverser...
- Tuples:aucune (un tuple est non modifiable)
- Ensembles : différence, intersection, union...
- Dictionnaires : appartenance, itération...

Unpacking

- Affecter le contenu d'un itérable dans des variables.
- Exemple :

```
taille = ["petit", "moyen", "grand"]

small, medium, large = taille
```

On peut forcer l'*unpacking* avec l'opérateur `*`

```
small, *autres = taille
```

Documenter

- Python intègre la documentation dans le code : Utilisation de commentaire par `#`
- Python permet du code autodocumenté pour les fonctions, classes, modules : utilisation après la première ligne. Ex. :

```
def f() :
   """
   Ceci est la doc de f()
   """
```

- La documentation sera accessible par l'attribut `__doc__`

Documenter (suite)

- `print(f.__doc__)`
- On a accès à la documentation via `help()` en mode interactif

Quelques modules standards

- `sys` fournit les paramètres et fonctions liés à l'environnement d'exécution,
- `string` fournit des opérations courantes sur les chaînes de caractères (équivalentes aux méthodes de la classe string),
- `os` fournit l'accès aux services génériques du système d'exploitation,
- `re` fournit le support des **expressions régulières** pour faire du pattern matching et des substitutions

Module `sys`

- `argv` : séquence des paramètres fournie au programme
- `argv[0]` contient le nom du programme
- `len(sys.argv)` fournit le nombre de paramètres passés
- `exit(val)` termine le programme en retournant `val`

Module **os**

- Permet d'accéder aux fonctions de l'OS et faire de la programmation système. Il est constitué de plusieurs sous-modules :
- `path`: manipulation de chemins
- `glob fnmatch` recherche de motifs sur nom de fichiers et répertoires
- `time` accès et manipulation de l'heure,
- `getpass` manipulation de mots de passe et logins

Module **`os.path`**

- `basename, dirname` : extraire nom fichier ou nom répertoire
- `split, join` : découpe ou construit un chemin
- `exists, isdir, isfile` : teste l'existence, le fait que ce soit un répertoire ou un fichier

Traitement des erreurs : les exceptions

- Lorsqu'un erreur survient dans un programme il est souhaitable de la traiter et non d'arrêter le programme.
- Une exception est un objet qui contient des informations sur le contexte de l'erreur.
- Exemple, comment traiter :

```
x = 1000
y = 0
z = x / y
```

Exemple

```
x = 1000
y = 0
try:
    z = x / y
except ZeroDivisionError:
    print(f"Une division par zéro
          est interdite z = {x} / {y}")
else:
    print(f"z = {x} / {y}")
```

Fonctionnement des exceptions

- Premièrement, les **instructions** placées **entre** les mots-clés **`try`** et **`except`** sont **exécutées**.
- **si aucune exception** n'intervient, la clause **`except`** est **sautée** et l'exécution de l'instruction `try` est terminée.
- **si une exception intervient** pendant l'exécution de la clause **`try`**, le **reste de cette clause `try` est sauté**. Si le type d'exception levée correspond à un nom indiqué après le mot-clé `except`, la clause `except` correspondante est exécutée, puis l'exécution continue après le bloc `try/except`.
- **si une exception intervient et ne correspond à aucune exception** mentionnée dans la clause except, elle est **transmise** à l'instruction **`try` de niveau supérieur** ; si aucun gestionnaire d'exception n'est trouvé, il s'agit d'une exception non gérée et **l'exécution s'arrête** avec un message.

Exemple avec différents types d'exception

```
fichier = '/tmp/produit.txt'
  try:
      fd = open(fichier)
  except PermissionError:
      print(f"Vous n'avez pas les droits sur {fichier}")
  except FileNotFoundError:
      print(f"Le fichier {fichier} n'existe pas")
```

Pour aller plus loin

- http://docs.python.org/tut/tut.html
- https://docs.python.org/3.8/contents.html
- docs.python.org
- Dive into Python

Les mots clés de Python

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

Quelques aphorismes de Python

```
>>> import this
The Zen of Python, by Tim Peters


Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```