Informatique

15

Aide Python

Cours

Aide Python

Nous voilà arrivés au bout de la découverte des outils qui permettront de répondre à tous les objectifs de l'enseignement de l'IPT en CPGE. Vous avez toutefois accès à bien plus, et pour apprendre à utiliser d'autres fonctions, vous aurez deux solutions :

- La plus simple consiste à trouver sur internet un code qui fait déjà ce que vous souhaitez
- La plus logique consiste à consulter l'aide proposée par Python

Regardez ce qui est écrit quand vous lancez pyzo :

```
Type 'help' for help, type '?' for a list of
*magic* commands.

>>> 
```

1.I. Aide sur une fonction

Pour obtenir l'aide sur une fonction, taper « help(nom de la fonction) ». Selon votre version de Python/Pyzo, essayer avec ou sans guillemets autour du nom de la fonction et avec ou sans parenthèses à la fin de la fonction.

Voici un extrait d'aide que l'on obtient en tapant les lignes :

```
from matplotlib import pyplot as plt
help(plt.plot)
```

```
plot(*args, **kwargs)
    Plot lines and/or markers to the
    :class:`~matplotlib.axes.Axes`.  *args* is a variable length
    argument, allowing for multiple *x*, *y* pairs with an
    optional format string.  For example, each of the following is
    legal::

        plot(x, y)        # plot x and y using default line style and color
        plot(x, y, 'bo')  # plot x and y using blue circle markers
        plot(y)           # plot y using x as index array 0..N-1
        plot(y, 'r+')     # ditto, but with red plusses

    If *x* and/or *y* is 2-dimensional, then the corresponding columns
    will be plotted.

    If used with labeled data, make sure that the color spec is not
    included as an element in data, as otherwise the last case
    ``plot("v","r", data={"v":..., "r":...)``
    can be interpreted as the first case which would do ``plot(v, r)``
    using the default line style and color.

    If not used with labeled data (i.e., without a data argument),
    an arbitrary number of *x*, *y*, *fmt* groups can be specified, as in::

        a.plot(x1, y1, 'g^', x2, y2, 'g-')

    Return value is a list of lines that were added.

    By default, each line is assigned a different style specified by a
    'style cycle'.  To change this behavior, you can edit the
    axes.prop_cycle rcParam.

    The following format string characters are accepted to control
    the line style or marker:
    ================    ===============================
    character           description
    ================    ===============================
    ``'-'``             solid line style
    ``'--'``            dashed line style
    ``'-.'``            dash-dot line style
    ``':'``             dotted line style
    ``'.'``             point marker
    ``','``             pixel marker
    ``'o'``             circle marker
                        [Liste volontairement raccourcie]
    ================    ===============================
    The following color abbreviations are supported:
    ==========  ========
    character   color
    ==========  ========
    'b'         blue
    'g'         green
    'r'         red
    'c'         cyan
    'm'         magenta
    'y'         yellow
    'k'         black
    'w'         white
    ==========  ========
    In addition, you can specify colors in many weird and
    wonderful ways, including full names (``'green'``), hex
    strings (``'#008000'``), RGB or RGBA tuples (``(0,1,0,1)``) or
    grayscale intensities as a string (``'0.8'``).  Of these, the
    string specifications can be used in place of a ``fmt`` group,
    but the tuple forms can be used only as ``kwargs``.

    Line styles and colors are combined in a single format string, as in
    ``'bo'`` for blue circles.

    The *kwargs* can be used to set line properties (any property that has
    a ``set_*`` method).  You can use this to set a line label (for auto
    legends), linewidth, anitialising, marker face color, etc.  Here is an
    example::
        plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
        plot([1,2,3], [1,4,9], 'rs',  label='line 2')
        axis([0, 4, 0, 10])
        legend()
[Fin de l'aide volontairement supprimée]
```

Sans rentrer dans tous les détails… Il faut évidemment déjà savoir lire, **de l'anglais en plus**. On voit ensuite dans l'exemple de « plot » ci-dessus que l'on peut donner des arguments de deux types différents :

- *arg : arguments à entrer les uns après les autres comme proposé dans l'aide (liste des x, des y, et en une seule chaine de caractères, les particularités du tracé (style de ligne et couleur)
- **kwargs : pour « keywords », ce sont des arguments associés à des mots clés prédéfinis comme « label » ou « linewidth » que l'on pourra modifier en écrivant le mot clé, « = » et la valeur voulue

Remarque : Il est important de regarder dans l'aide obtenue (qui est parfois difficile à déchiffrer) :

- La syntaxe pour l'appel de la fonction : 1ère ligne
- Ce que va renvoyer/réaliser la fonction : 2ème ligne en général
- Le paragraphe "parameters" qui précise les types des différents paramètres, et les paramètres qui sont éventuellement optionnels (cf numpy.arange).

Remarque : quand il n'y en a pas « trop », les paramètres sont précisés dans la première ligne, et les optionnels sont mis entre crochets. Il ne faut pas pour autant mettre les crochets si on souhaite les définir (cf math.log) :

```
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.
```

1.II. Le ?

Ecrire « ? + fonction » permet d'en savoir plus sur les commandes pré-existantes de python (pas d'import de librairie). Vous voulez en savoir plus sur les listes, input, max, print… Ecrivez « ? + commande » et faites entrée :

```
>>> ? max
max(iterable, *[, default=obj, key=func]) ->
value
max(arg1, arg2, *args, *[, key=func]) -> val
ue


>>> ? list
Built-in mutable sequence.

If no argument is given, the constructor cre
ates a new empty list.
The argument must be an iterable if specifie
d.
```

1.III. Liste de méthodes (fonctions) prédéfinies associées à des objets

Pour obtenir toutes les méthodes prédéfinies dans python, on peut utiliser « dir ». Par exemple :

- « dir(list) » donnera toutes les méthodes utilisables sur les objets de type list
- « dir(math) » permettra d'obtenir la liste de toutes méthodes présentes dans le module math (après l'avoir importé)

Remarque : les méthodes « __methode__ » sont appelées « magic methods » (cf programmation objet) et permettent de définir des choses comme ce qu'il faut afficher quand on tape L (une liste) puis « enter » dans la console, ou de définir ce qu'il faut faire quand on somme deux listes… Elles sont transparentes pour nous. Les autres méthodes s'utilisent avec un point « . ». Par exemple, si L est une liste, L.append() ajoute un terme. La méthode append est une fonction des objets de type liste.

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribut
e__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__
init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__
', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__
', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'ind
ex', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> import math

>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'ac
os', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'co
pysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1
', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', '
gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldex
p', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

On pourra aller plus loin avec help, par exemple :

```
>>> help(list.append)
Help on method_descriptor:

append(...)
    L.append(object) -> None -- append object to end


>>> import math

>>> help(math.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.
```