Python IV: Módulos predefinidos

Comandos mágicos de IPython

Antes de adentrarnos en las profundidades de Python, introduciremos los *comandos mágicos* de IPython (ejecutables tanto en IPython como en Jupyter). Éstos pueden facilitarnos el trabajo de distintas maneras. Pueden encontrar la lista completa de comandos mágicos aquí; abajo revisamos algunos que nos serán útiles en este curso. Los comandos mágicos se definen con un `%` al comienzo. Por ejemplo:

`%lsmagic` - imprime todos los comandos mágicos disponibles

`%history` - Imprime la historia reciente de comandos

`%run` - ejecuta un programa escrito en python dentro de IPython

`%sx` - ejecuta un comando shell y entrega el resultado

`%time` - mide cuánto tiempo toma ejecutar una operación

`%timeit` - como %time, pero ejecuta el código tantas veces como convenga según la duración del comando, para obtener una medición más robusta

`%who` - muestra todas las variables, opcionalmente de un cierto tipo

Estos son comandos mágicos *de línea* - operan sobre el contenido de la misma línea en la que se ejecutan. Existen también los comandos mágicos *de celda*, que operan sobre toda la celda.

`%%bash` - ejecuta la celda usando bash (cuando está disponible)

`%%time / %%timeit` - análogos a sus versiones de línea pero para una celda completa

Con este último podemos demostrar por ejemplo la diferencia entre definir una lista usando un ciclo for y usando comprensión de lista:

```
n  = 10000
```

```
%%timeit
```

```
x = []
```

```
for i in range(n):
```

```
    x.append(i**0.5)
```

```
> 1.02 ms ± 22.9 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

```
%%timeit
```

```
x = [i**0.5 for i in range(n)]
```

```
> 774 µs ± 7.2 µs per loop (mean ± std. dev. of 7 runs,
1000 loops each)
```

Una mejora de un 25%, además de haber reemplazado tres líneas de código por una sola.

Si solo fuéramos a usar cada elemento una vez, entonces convendría definirlo como un generador:

```
%%timeit
```

```
x = (i**0.5 for i in range(n))
```

```
> 399 ns ± 26.7 ns per loop (mean ± std. dev. of 7 runs,
1000000 loops each)
```

Este último es *mil* veces más rápido!

Hecho esto podemos verificar las variables definidas:

```
%who
```

```
> NamespaceMagics    get_ipython    json      n    sys  x
```

Como último ejemplo, si tenemos un archivo test.py:

```
with open('test.py', 'w') as f:
```

```
    print('print("Saludo de prueba desde test.py")',
file=f)
```

Podemos ejecutarlo desde Jupyter o IPython con

```
%run test.py
```

```
> Saludo de prueba desde test.py
```

O incluso código en bash:

```
%%bash

wc test.py

> 1  5 32 test.py
```

Módulos predefinidos

Python incluye un gran número de módulos predefinidos - esto es, que están disponibles con la instalación base de Python. Para incluirnos en nuestro programa debemos *importarlos*. La sintaxis es

```
import modulo
```

o, para importar solo parte específica de un módulo:

```
from modulo import submodulo

from modulo.submodulo import subsubmodulo

...
```

`sys` - Información y utilidades de sistema

```
import sys

sys.exec_prefix

> 'C:\\Users\\cjsif\\Miniconda3'

sys.executable

> 'C:\\Users\\cjsif\\Miniconda3\\python.exe'

sys.platform

> 'win32'
```

Podemos leer la variable de sistema **`PATH`**:

```
sys.path

> ['C:\\Users\\cjsif',

...

'C:\\Users\\cjsif\\.ipython']
```

En Windows podemos hacer:

```
sys.getwindowsversion()

> sys.getwindowsversion(major=10, minor=0, build=18362,
platform=2, service_pack='')
```

También podemos conocer la versión de Python que estamos usando:

```
sys.version_info

> sys.version_info(major=3, minor=8, micro=3,
releaselevel='final', serial=0)
```

Podemos obtener el tamaño de un objeto en la memoria:

```
x = range(10000)

sys.getsizeof(x)

> 48

x = list(x)

sys.getsizeof(x)

> 80056
```

Aquí queda demostrada la diferencia entre listas y generadores a la que nos hemos referido. Es importante tener en cuenta que `getsizeof` no calcula el tamaño en memoria de manera recursiva, sino sólo superficialmente:

```
x[0] = x

sys.getsizeof(x)

> 80056
```

Para terminar la ejecución de un programa:

```
sys.exit()
```

Podemos usar este módulo para leer información entregada desde la terminal a un programa de Python:

```
with open('test_args.py', 'w') as f:
     print('import sys', file=f)
     print('args = sys.argv', file=f)
     print('print(f"Recibi {len(args)} argumentos:
{args}")', file=f)
```

```
        print('print(f"El último de ellos es {args[-1]}")',
    file=f)
```

Luego desde la terminal:

```
$ python test_args.py primero segundo

Recibi 3 argumentos: ['test_args.py', 'primero',
'segundo']

El último de ellos es segundo
```

O desde Jupyter/IPython:

```
%run test_args.py primero segundo
```

`os` - Información y utilidades del sistema operativo

Podemos usar este módulo para acceder a las variables de entorno que vimos con bash:

```
import os

os.environ

> ...
```

Así, por ejemplo

```
os.environ.get('HOME', os.environ.get('HOMEPATH'))

> 'C:\\Users\\cjsif'
```

Como ya hemos visto, otra manera de asegurar un resultado sería

```
homekey = 'HOME' if 'HOME' in os.environ else 'HOMEPATH'

os.environ.get(homekey)

> 'C:\\Users\\cjsif'
```

y al directorio actual:

```
os.getcwd()

> 'C:\\Users\\cjsif\\Documents'
```

para ver el contenido de un directorio:

```
os.listdir('./')
```

```
> ['.ipynb_checkpoints', 'Untitled.ipynb']
```

(el módulo `glob` provee una versión más versátil de `os.listdir`) para cambiar de directorio:

```
os.chdir('..')

os.getcwd()

> 'C:\\Users\\cjsif'
```

y para manipular archivos:

```
os.mkdir('nuevo_dir')

os.makedirs('nuevo_dir_profundo/subdir')
```

Si el directorio existe se genera un `FileExistsError`, excepto si agregamos el argumento `exist_ok`:

```
os.makedirs('nuevo_dir_profundo/subdir', exist_ok=True)
```

Podemos eliminar un archivo:

```
os.remove('test.txt')
```

o un directorio *vacío*:

```
os.rmdir('nuevo_dir')
```

De hecho, podemos acceder a la terminal y ejecutar cualquier comando válido en bash:

```
os.system('touch test2.txt')
```

Para mejores utilidades de manejo de archivos, revisar **`shutil`**.

`time`

```
import time

time.time()

> 1599658003.9643712

time.ctime()

> 'Wed Sep  9 10:25:45 2020'
```

```
time.localtime()

>  time.struct_time(tm_year=2020,  tm_mon=9,  tm_mday=9,
tm_hour=10, tm_min=29, tm_sec=8, tm_wday=2, tm_yday=253,
tm_isdst=1)

time.localtime().tm_yday

> 253

time.strftime('%Y-%m-%d %H:%M', time.localtime())

> '2020-09-09 10:36'
```

También podemos usar este módulo para saber cuánto demora nuestro programa en correr:

```
t0 = time.time()

x = [i**0.5 for i in range(10000)]

t1 = time.time()

print(f'Nos demoramos {1e3*(t1-t0):.1f} ms')
```

El módulo `datetime` entrega mayores utilidades para trabajar con fechas y horarios.

Módulos para trabajo numérico

`math`

```
import math

math.pi

> 3.141592653589793

math.floor(3.84)

> 3

math.ceil(3.84)

> 4

math.sqrt(9)
```

```
> 3.0

math.prod(range(1, 5))

> 24

math.factorial(4)

> 24

math.cos(2*math.pi)

> 1.0

math.log(math.e)

> 1.0

math.log10(10)

> 1.0

math.exp(1)

> 2.718281828459045

math.hypot(3, 4)

> 5.0
```

También podemos por ejemplo evaluar si un número es finito:

```
math.isfinite(3.84)

> True

math.isfinite(math.inf)

> False
```

o si está definido. Un número indefinido es representado por `NaN` o `nan` ("Not a Number"). Algunas de las operaciones que generan `NaN` son:

```
math.inf - math.inf

> nan

0*math.inf

> nan
```

```
math.isnan(3.84)

> False

math.isnan(math.inf/math.inf)

> True
```

De hecho, hay algunas funciones matemáticas incorporadas en el espacio base, sin necesidad de importar ningún módulo:

```
abs(-3.84)

> 3.84

max(range(5))

> 5

min(range(5))

> 0

round(3.84)

> 4

sum(range(5))

> 10
```

Pueden encontrar mucha mayor funcionalidad en la documentación oficial, https://docs.python.org/3/library/math.html

random

```
import random

random.random()

> 0.7773647927372435

random.randrange(10)

> 3
```

Es lo mismo que

```
random.randint(0, 9)
```

```
> 9
```

Para números reales:

```
random.uniform(0, 10)

> 6.894377936418481
```

Nota importante: `random`, como todos los generadores de números aleatorios, es en realidad un generador de números *seudo-aleatorios*. Son aleatorios en el sentido que un conjunto de valores es estadísticamente aleatorio, pero seudo-aleatorios en el sentido que la secuencia depende completamente de la **semilla inicial** del generador:

```
random.range(10)

> 5

random.seed(6)

random.random()

> 0.793340083761663

random.seed(6)

random.random()

> 0.793340083761663
```

Podemos generar números aleatorios a partir de algunas distribuciones de probabilidad comunes (como la distribución uniforme de arriba):

```
random.gauss(0, 1)

> -1.1788417512306717

random.expovariate(3)

> 0.5285633227246923
```

Por último, podemos elegir un valor aleatorio a partir de una secuencia de números predefinida:

```
x = [0, 0, 4, 8, 2]

random.choice(x)

> 2
```

```
random.choice(x)

> 0
```

etc.

Un último módulo predefinido

`argparse`

El módulo argparse permite leer argumentos desde la terminal de manera más organizada que `sys.argv`, aunque toma un poco más de trabajo:

```
import argparse
```

Lo primero que debemos hacer es definir un *objeto* `ArgumentParser`, al que opcionalmente podemos darle una descripción:

```
parser = argparse.ArgumentParser(

    description='Programa simple')
```

Luego debemos definir los argumentos que queremos permitir, usando la función `add_argument`:

```
parser.add_argument(

    'numero', type=int, help='Numero entero')
```

Una vez agregados todos los argumentos (veremos más opciones en un segundo), debemos interpretarlos:

```
parser.parse_args()
```

Este ejemplo simple está guardado en el archivo `ejemplo_argparse_1.py`, que además tiene un mensaje sencillo que imprime el número recibido. Ahora para ejecutar este archivo *debemos* entregar el argumento requerido; si no lo hacemos el error será informativo (a diferencia de `sys.argv` que solo generaría un `IndexError`):

```
$ python ejemplo_argparse_1.py

usage: ejemplo_argparse.py [-h] numero

ejemplo_argparse.py: error: the following arguments are
required: numero
```

En cambio,

```
$ python ejemplo_argparse_1.py 3

Recibi el numero 3

El cuadrado de 3 es 9
```

Esto demuestra que el número recibido es interpretado como un entero inmediatamente (gracias al argumento `type=int` de arriba). La otra gran ventaja de argparse es que automáticamente genera una opción de ayuda con información de uso:

```
$ python ejemplo_argparse_1.py -h

usage: ejemplo_argparse.py [-h] numero

Programa simple

positional arguments:

  numero  Numero entero

optional arguments:

  -h, --help  show this help message and exit
```

Podemos agregar argumentos opcionales anteponiendo guiones, tal como en Bash:

```
parser.add_argument('--minimo', type=int, default=0)

parser.add_argument('--maximo', type=int, default=100)
```

Este ejemplo está desarrollado en el archivo `ejemplo_argparse_2.py`, de manera que:

```
$ python ejemplo_argparse_2.py -h

usage: ejemplo_argparse_2.py [-h] [--minimo MINIMO]
[--maximo MAXIMO] numero

Programa con argumentos opcionales

positional arguments:

  numero       numero entero

optional arguments:

  -h, --help   show this help message and exit
```

```
  --minimo MINIMO

  --maximo MAXIMO

$ python ejemplo_argparse_2.py 12

El numero 12 esta en el rango [0,100]

$ python ejemplo_argparse_2.py --minimo 20 12

El numero 12 es menor que el minimo de 20
```

Además de especificar el tipo de argumento, podemos especificar la cantidad de valores esperados para un argumento (`ejemplo_argparse_3.py`):

```
parser.add_argument(

    '-r', '--rango', type=int, nargs=2, default=[0,100])
```

Podemos especificar “al menos un argumento” (`nargs='+'`) u “opcionalmente al menos un argumento” (`nargs='*'`), como muestra `ejemplo_argparse_4.py`:

```
add_arg = parser.add_argument

add_arg(

    '--multiplicar', type=int, nargs='+',

    help='lista de numeros para multiplicar')

add_arg(

    '--para-sumar', type=int, nargs='*',

    default=range(1,11), help='numeros para sumar')
```

De manera que

```
$  python ejemplo_argparse_4.py -h

usage:    ejemplo_argparse_4.py    [-h]    [--multiplicar
MULTIPLICAR [MULTIPLICAR ...]] [--para-sumar [PARA_SUMAR
[PARA_SUMAR ...]]]

Programa con variaciones de nargs

optional arguments:

  -h, --help        show this help message and exit

  --multiplicar MULTIPLICAR [MULTIPLICAR ...]
```

```
                        lista de numeros para multiplicar
  --para-sumar [PARA_SUMAR [PARA_SUMAR ...]]
                        numeros para sumar
```

Último ejemplo: podemos pedir también argumentos booleanos (`ejemplo_argparse_5.py`):

```
add_arg('-v', '--verbose', action='store_true')
```

La ayuda de este argumento y la ejecución del programa se ven así:

```
$  python ejemplo_argparse_5.py -h
usage: ejemplo_argparse_5.py [-h] [-v]
Ejemplo de argumento booleano
optional arguments:
  -h, --help   show this help message and exit
  -v, --verbose
$ python ejemplo_argparse_5.py
Verbosidad False
$ python ejemplo_argparse_5.py -v
Verbosidad True
```