

Intérprete de EXE → `#!/usr/bin/env python`
`#_*_coding: utf8`

`print()` → Print normal, podemos añadir `.format()` para que sea como un printf

```
print("Hola {} tienes {} años".format(nombre, edad))
```

if a **in** b → Si hay algo de a en b

`int(a)` → Identificador de identidad, verifica que la variable sea int

`input()` → Entrada de datos de un usuario

`for m in range(0, variable):` → Ejemplo estructura de programación en este lenguaje se puede obviar los parentesis y hacer mezclas un tanto farragosas

LISTAS, TUPLAS, DICCIONARIOS

*Lista → Se puede alterar y hacer lo que queramos como un `pop()`

*Tupla → Lista que no se puede alterar

*Diccionario → La estructura de `{'key': value}`

Ejemplo de hacer una lista un poco enrevesada:

```
lista= [i for i in range(0,10000) if i%2==0]
```

En python se pueden hacer guarradas de este estilo.

LECTURA/ESCRITURA FICHEROS

`archivo= open('./archivo.txt')` → Para instanciar un archivo, si no existe lo crea

`archivo.write("Bla bla bla")` → Para escribir en el fichero

`archivo.read()` → Lee un carácter

`archivo.readlines()` → Lee todas las líneas

Ejemplo útil:

```
for l in archivo.read().split("\n"):
    print(l)
```

`archivo.close()` → cierra el archivo y deja de tenerlo en memoria

`import os`

`os.system("cls")` → Para usar comandos de consola desde python como por ejemplo limpiar la pantalla

`os.Path` → Es la gestión de ficheros

`path.exists("ruta fichero")` → dice si existe el fichero

GESTIÓN DE FUNCIONES Y CLASES

`def main():` → Para definir una función cualquiera

`pass` → Para decir que ya lo harás luego

`__name__` → Dice el nombre de la función actual

`if __name__ == "__main__":` → Para ejecutar la función main()
`main()`

Para crear una clase:

`class Coche(object):`

`def __init__(self):` → Función inicial del objeto autoejecutable

`self.modelo="Fiat 500"` → Variable del objeto

DECORADORES

@classmethod → Hace que exista una función dentro del objeto y desvinculada del `__init__`, podemos invocar la con parámetros distintos, solo que hay que pasarle el argumento "cls"

```
@classmethod
def saludo(cls, nombre):
    print("Hola {}".format(nombre))
```

@staticmethod → Igual que classmethod pero no depende de ningún parámetro ni de instancia ni nada, no se le pasa NINGÚN

```
@staticmethod
def despedida():
    print("Hasta luego")
```

ERRORES

Aquí `try -- catch` se convierte en **try -- except**

Tipos de errores:

***NameError** → Errores de variables, lógicos

***KeyboardInterrupt** → Para errores de interrupción del programa

raise Exception → Para generar un Error

Para los errores y su gestión tenemos el módulo **logging**

LA F

Es una forma de concatenar texto muy limpia

`f'Hola esto {es} un texto de {variable}'` → Metemos dentro de {} una variable y automáticamente se concatena

Función `callback(Autoejecutable)`:

```
yield response.follow(url=link, callback= self.parse_country)
```

WEB SCRAPING



TEORÍA DEL SCRAPEO Y TAL

Scraping es la extracción de datos de un sitio Web, para realizar dicho proceso lo segmentamos en distintas tareas:

+Spiders → Un spider es el programa encargado de extraer la información del sitio web

+Middlewares → Sirve para gestionar las peticiones request/response, inyección de cabeceras y gestionado de proxys.

+Pipelines → El pipeline es el código que refina los datos extraídos, se encarga de eliminar duplicados, refinar el código y almacenarlo en una BD.

+Engine → Es el motor que se asegura que todas las operaciones salgan bien (main)

+Scheduler → Es el planificador que se encarga de coordinar los procesos, se asegura que las requests/responses salgan bien.

El archivo **Robots.txt** es un archivo que se añade en la raíz de un sitio web para que limite a los Spiders y hacer la información pública o privada.

El archivo tiene 3 campos:

***User-Agent** → Dice a que robots se aplican las normas (mozilla, chorme...)

***Disallow** → Deshabilita todos los directorios señalados a los spiders

***Allow** → Habilita los sitios indicados (es el valor por defecto)

Ejemplo (todos los robots NO pueden entrar a...):

```
User-agent: *  
Disallow: /topsy/  
Disallow: /crets/  
Disallow: /hidden/file.html
```

EXPRESIONES XPATH

Una expresión XPATH es un buscador, como una URL o una expresión regular, pero para ficheros, una jerarquía.

Al igual que las rutas, para programar es mejor usar un XPATH relativo ya que nos hará el código mucho más flexible.

Ejemplo XPATH → `S:\GitHubFull\GitHub-Python\`

Como hacer XPATH:

Suponemos que existe un árbol (a-b(b.1, b.2, b.3)-c-d)

`*"/a"` → Barra al principio indica hijo de... , sinó es directorio raíz

`*//b` → Indica descendiente y que viene de un padre, es válido `/a//b`

`*../` → Marcha atrás, es decir `/a//b/.. == /a`

`*a/b/b.1|b.2|b.3` → marca distintas rutas simultáneas.

Predicados

`*/a/b[@b1]` → `[@algo]` se pone al lado de alguno para que se muestre solo si tiene ese descendiente

`*a/b[1]` → Selecciona el primer elemento dentro de b, sería b.1

`*a/b[last()-1]` → Selecciona el último menos uno (b.2)

Condiciones

`*/a/b/b.1[atributo > 10]` → Selecciona al "atributo", si cumple la condición

`*a/b/b.1[. > 10]` → Selecciona a el elemento b.1 si es mayor a 10

`*a[b=10 and c=100]` → Se pueden hacer selecciones compuestas

Funciones

`*a/node()` → Selecciona todo de todos los nodos

`*a/b/text()` → Selecciona únicamente el texto del nodo

`*a/b//text()` → Selecciona el texto del nodo y de los descendientes

`*/a/*` → Selecciona todos los elementos

`*a/@` → Selecciona todos los atributos del nodo

`*a//@` → Selecciona todos los atributos de solo el padre

`*a/b[@class="claseB"]` → Para seleccionar un atributo de B y le metemos condición

El programa usado para hacer scraping será Anaconda

Para instalar un paquete usaremos el comando **conda install *paquete***

Usaremos: scrapy, ipython (mayor funcionalidad a la terminal de python)

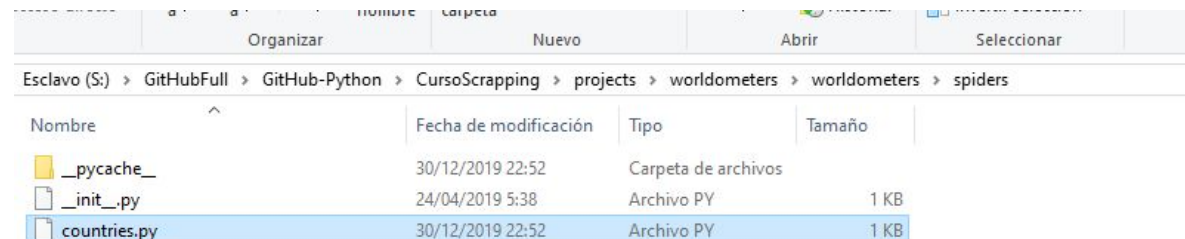
Con el comando scrapy podemos crear un nuevo proyecto, que nos dará todos los elementos mencionados(spiders, pipelines...)

CREAR UN SPIDER

Para crear un Spider iremos en la terminal al directorio de trabajo y ejecutaremos:

***scrapy genspider nombreSpider url** → Creamos un spider de un sitio web, **IMPORTANTE** no poner el https ni la última / de la url

Y se creará en nuestra carpeta de Spiders:



The screenshot shows a Windows File Explorer window with the address bar displaying the path: Esclavo (S:) > GitHubFull > GitHub-Python > CursoScrapping > projects > worldometers > worldometers > spiders. The main area shows a list of files and folders:

Nombre	Fecha de modificación	Tipo	Tamaño
__pycache__	30/12/2019 22:52	Carpeta de archivos	
__init__.py	24/04/2019 5:38	Archivo PY	1 KB
countries.py	30/12/2019 22:52	Archivo PY	1 KB

TERMINAL: SCRAPY


Comando Scrapy


```
(virtual_enviroment) S:\GitHubFull\GitHub-Python\CursoScrapping\projects\worldometers>scrapy
Scrapy 1.6.0 - project: worldometers
Usage:
  scrapy <command> [options] [args]


Available commands:
bench          Run quick benchmark test
check          Check spider contracts
crawl          Run a spider
edit           Edit spider
fetch          Fetch a URL using the Scrapy downloader
genspider      Generate new spider using pre-defined templates
list           List available spiders
parse          Parse URL (using its spider) and print the results
runspider      Run a self-contained spider (without creating a project)
settings       Get settings values
shell          Interactive scraping console
startproject   Create new project
version        Print Scrapy version
view           Open URL in browser, as seen by Scrapy

Use "scrapy <command> -h" to see more info about a command

(virtual_enviroment) S:\GitHubFull\GitHub-Python\CursoScrapping\projects\worldometers>
```

 Proyecto

 Para testeo y uso de spiders manual

 Para generar un nuevo proyecto

Creamos un proyecto y nos metemos en el Shell, para ir testeando

fetch() → Para abrir spiders si nos podemos conectar (200 ok, 404 not found), una vez hecho fetch exitoso podemos hacer, si queremos ver la respuesta, ponemos el comando **response.body**, que será el cuerpo entero

```
In [11]: fetch("https://www.worldometers.info/world-population/population-by-country/")
2019-12-31 14:09:43 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.worldometers.info/world-population/population-by-country/> (referer: None)
In [12]: .
```

Podemos instanciar una variable

r= scrapy.Request("URL") → Recogemos el GET

view(response) → abre en el navegador la response

response.xpath("//h1/text()) → Devuelve el texto que contenía el nodo h1.

response.css("selector css") → Igual que el de arriba pero con css

response.xpath("//h1/text()).get() → Quita la paja y te da directamente eso PERO SOLO DE UNO

response.xpath("//h1/text()).getall() → Da directamente el texto, pero devuelve un array en el que están todos los elementos.

IMPORTANTE DESACTIVAR JS (Ctrl+P + disable JS + Ctrl+R) el spider no lo renderiza

EJECUTAR CRAWL/SPIDER

Con el spider creado:

```
import scrapy

class CountriesSpider(scrapy.Spider):
    #Identificador Único en todo el proyecto
    name = 'countries'
    allowed_domains = ['www.worldometers.info/world-population/population-by-country']
    start_urls = ['https://www.worldometers.info/world-population/population-by-country/']

    def parse(self, response):
        #El titulo de la página entera
        title= response.xpath("//h1/text()").get()
        #Todos los paises
        countrie= resposne.xpath("//td/a/text()").getall()

        yield{
            "title": title,
            "countries": countrie,
        }
```

Nos dirigimos al directorio en la terminal que contenga el fichero de configuración (scrapy.cfg)

`scrapy crawl id` → Siendo la id el name del fichero del spider

SPIDER MULTIPAGE

Una vez creado el fichero del Spider crearemos su estructura con los comandos que ejecutamos en la terminal.

Pero hay un par de cosas que hay que saber

```
class CountriesSpider(scrapy.Spider):
    name = 'countries'
    allowed_domains = ['www.worldometers.info']
    start_urls = ['http://www.worldometers.info/world-population/population-by-country/']

    def parse(self, response):
        countries= response.xpath("//td/a")
        for country in countries:
            name= country.xpath("./text()").get()
            link= country.xpath("./@href").get()

            yield response.follow(url=link, callback= self.parse_country, meta= {"nombre_pais": name})

#Segunda página web
def parse_country(self, response):
    rows= response.xpath("//table[@class='table table-striped table-bordered table-hover table-condensed table-list'])[1]")
    nombre= response.request.meta["nombre_pais"]
    for row in rows:
        year= row.xpath("./tr/td[1]/text()").get()
        population= row.xpath("./tr/td[2]/strong/text()").get()
        yield{
            "Nombre del pais": nombre,
            "Año": year,
            "Poblacion": population
        }
```

Arriba el ejemplo perfecto.

-yield response.follow(url= link, callback= self.funcionPag2, meta={"dato": 10}) → Esta función concatena nuestra url actual con la que hayamos metido en la variable "link", callback llama a otra función para controlar la segunda página con sus datos respectivos y metemos en el meta los datos que queramos usar en la pág 2 ya que sinó se pierden

-valorDePagAnterior= response.request.meta["dato"] → Para recoger desde la segunda página el dato

scrapy crawl *nombreSpider* -o datos.json → Para exportar todo lo que nos ofrece nuestro spider a un fichero externo, para persistir la información

Alt+Shift+F → Formatear JSON generado

MENÚ EN TERMINAL PREHECHO

import **argparse** → Librería que nos proporciona una interfaz para comunicarnos con la terminal a nuestro programa.

parser= **argparse.ArgumentParser**(description="De que va el programa") →
Se crea la variable y le decimos de que se va a tratar nuestro menú.

`parser.add_argument("-t", "--tusMuertos", help="Variable tus muertos")` →

Creamos un parámetro, ponemos -t + un string por consola, podemos crear tantos como queramos, para luego acceder con `parser.tusMuertos` y siendo una variable instanciada.

`parser= parser.parse_args()` → Para añadir al menú todas las variables.

CONEXIÓN A PÁGINAS WEB

GET

`import requests` → Librería que nos ayuda a conectarnos a sitios web

`url = requests.get(url="https://google.es")` → Para obtener una requests de Google

Podemos pasarlo a un diccionario para leerlo mejor

```
url= requests.get(url= parser.target)
cabeceras = dict(url.headers)
for x in cabeceras:
    #Para recorrer el diccionario
    print(x + " : " + cabeceras[x])
```

POST

Nota: `request.text` → Nos da el cuerpo del HTML para un String