



ENTRADA Y SALIDA DE FICHEROS

PROGRAMACIÓN III

Abdel G. Martínez L.

CONCEPTO Y DEFINICIÓN

- ⦿ Hasta el momento toda entrada de datos ha sido a través del teclado y toda salida de datos, en pantalla
- ⦿ Python provee funcionalidades básicas e integradas para manipular objetos de ficheros
- ⦿ No se necesita importar ninguna librería

ABRIR UN FICHERO

- ⦿ Antes de poder leer o escribir en un archivo, debemos abrirlo
- ⦿ Python implementa la función `open()`
- ⦿ Es una función que retorna un objeto fichero
- ⦿ Utiliza dos argumentos:
 - *Nombre del fichero*
 - *Modo de apertura*

```
>>> archivo = open('data.txt', 'w')
```

MODOS DE APERTURA

Texto	Descripción
r	Sólo lectura
r+	Lectura y escritura
w	Sólo escritura
w+	Lectura y escritura
a	Adjuntar
a+	Adjuntar y lectura

Binario	Descripción
rb	Sólo lectura
rb+	Lectura y escritura
wb	Sólo escritura
wb+	Lectura y escritura
ab	Adjuntar
ab+	Adjuntar y lectura

CREAR UN FICHERO

- ⦿ En el siguiente ejemplo crearemos un fichero llamado `prueba.txt`

```
>>> archivo = open('prueba.txt', 'w')
```

```
>>> archivo.write("Nuevo archivo\n")
```

```
>>> archivo.write("de dos lineas\n")
```

```
>>> archivo.close()
```

LEER UN FICHERO

- ⦿ En el siguiente ejemplo leeremos el fichero llamado `prueba.txt`

```
>>> archivo = open('prueba.txt', 'r')
```

```
>>> print(archivo.read())
```

- ⦿ Ahora, los 5 primeros caracteres de datos

```
>>> archivo = open('prueba.txt', 'r')
```

```
>>> print(archivo.read(5))
```

ITERAR SOBRE UN OBJETO DE FICHERO

- ⦿ Para leer cada línea de un fichero, podemos iterar

```
>>> archivo = open('prueba.txt', 'r')
```

```
>>> for linea in archivo:
```

```
...     print linea
```

CERRAR UN FICHERO

- ⦿ Cuando se termina de trabajar sobre un fichero, debemos cerrarlo
- ⦿ Esto libera recursos tomados al abrir el fichero
- ⦿ Cualquier posible acción, luego de cerrar el fichero, fallará
- ⦿ La función para cerrar un fichero es `close()`

```
>>> archivo = open('prueba.txt', 'r')
```

```
>>> archivo.close()
```



```

PPOptional.add_argument('-v', dest='verbose', action='count', default=0, help='increase verbosity level')
PPOptional.add_argument('--version', action='version', version='Version ' + str(constant.CURRENT_VERSION), help='laZagne version')

PWrite = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
PWrite._optionals.title = 'output'
PWrite.add_argument('-w', dest='write', action='store_true', help='write a text file on the current directory')

all_subparser = []
for c in category.keys():
    category[c]['parser'] = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
    category[c]['parser']._optionals.title = category[c]['help']

    category[c]['subparser'] = []
    for module in modules[c].keys():
        m = modules[c][module]
        category[c]['parser'].add_argument(m.options['command'], dest=m.options['dest'], action=m.options['action'], help=m.options['help'])

        if m.suboptions and m.name != 'thunderbolt':
            tmp = []
            for sub in m.suboptions:
                tmp_subparser = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
                tmp_subparser._optionals.title = sub['title']
                if 'type' in sub:
                    tmp_subparser.add_argument(sub['command'], dest=sub['dest'], action=sub['action'], help=sub['help'])
                else:
                    tmp_subparser.add_argument(sub['command'], dest=sub['dest'], action='store_true', help=sub['help'])
                tmp.append(tmp_subparser)
            all_subparser.append(tmp_subparser)
        category[c]['subparser'] += tmp

parents = [PPOptional] + all_subparser + [PWrite]
dic = {'all': {'parents': parents, 'help': 'Run all modules', 'func': runAllModules}}
for c in category.keys():
    parser_tab = [PPOptional, category[c]['parser']]
    if 'subparser' in category[c]:
        if category[c]['subparser']:
            parser_tab += category[c]['subparser']
    parser_tab += [PWrite]
    dic_tmp = {c: {'parents': parser_tab, 'help': 'Run %s module' % c, 'func': runModule}}
    dic = dict(dic.items() + dic_tmp.items())

subparsers = parser.add_subparsers(help='Choose a main command')
for d in dic.keys():
    subparsers.add_parser(d, parents=dic[d]['parents'], help=dic[d]['help']).set_defaults(func=dic[d]['func'], auditType=d)

```



MÓDULOS

PROGRAMACIÓN III
Abdel G. Martínez L.

CONCEPTO Y DEFINICIÓN

- ⦿ No desarrollemos aplicaciones complejas sin organizar en funciones
- ⦿ Un módulo es un grupo de funciones alojadas en un archivo `.py`
- ⦿ Es como un conjunto de herramientas que podemos usar siempre

EJEMPLO

- ◉ En un archivo llamado `finanzas.py` creamos la siguiente función

```
def calcularImpuesto(precio, impuesto):  
    total = precio + (precio * impuesto)  
    return total
```

IMPORTAR UN MÓDULO

⦿ Existen dos posibilidades:

- `import`

- Manera más simple y fácil de utilizar

- Importa todas las funciones dentro del módulo

```
import finanzas
```

IMPORTAR UN MÓDULO

⦿ Existen dos posibilidades:

- `from`

- Es más recomendable

- Ahorra tiempo de procesamiento

- Importa funciones específicas del módulo

```
from finanzas import calcularImpuesto
```

USO DE MÓDULOS

- ⦿ Una vez importados, la forma de utilizarlos es el nombre del módulo, un punto, el nombre de la función y los paréntesis (junto a sus argumentos)

```
finanzas.calcularImpuestos(50, 0.1)
```

MÓDULOS INCORPORADOS

- ⦿ Existen módulos ya preexistentes en Python
- ⦿ Son más de 60,000 disponibles
- ⦿ Los más importantes son:
 - `math`: Provee acceso a funciones y constantes matemáticas
 - `os`: Manipula comandos de sistema operativo
 - `datetime`: Contiene información de fechas y tiempos