Calendario

- Módulos
  - From Import
  - PYTHONPATH
  - __name__
- Paquetes
  - __init__

Módulos

- Podemos dividir nuestro programa en diferentes módulos (archivos .py).
- Los módulos son entidades que permiten una organización y división lógica de nuestro código
- Ejemplo: Creamos un archivo mi_modulo.py

```
def mi_funcion():
        print "una funcion"

class MiClase:
        def __init__(self):
                print "una clase"

print "un modulo"
```

- ¿Cómo hacemos si queremos utilizar la función o la clase en otro módulo? (por ahora estamos poniendo todo nuestro código en un mismo módulo o archivo)

Módulos (2)

- Si quisiéramos utilizar alguna funcionalidad definida en otro módulo en nuestro programa, tenemos que importarlo.
- Para importar un módulo se utiliza la palabra clave **import** seguida del nombre del módulo.
- Ejemplo:
  - import mi_modulo
  - Para invocar la función *mi_funcion()* defendida en *mi_modulo*, basta con escribir:
  - *mi_modulo.mi_funcion()*
- Tener en cuenta que el import ejecuta el código del módulo (es decir, si hay código suelto, fuera de una función o clase, se va a ejecutar)

Módulos (3)

- La clausula import también permite importar varios módulos en la misma línea
- os, sys, time son algunos de los módulos de la librería estandar.
  - *import os, sys, time*
  - *print (time.asctime())*
- Os:
  - funcionalidad relacionada con sistema operativo
- Sys:
  - funcionalidad relacionada con el intérprete de Python
- Time:
  - funciones para manipular fechas y horas

From-import

- Utilizando import es necesario preceder el nombre de la función o la clase con el nombre del módulo al que pertenecen (*time.asctime()*).
- Podemos utilizar la sentencia *from-import* para ahorrarnos el nombre del módulo.
- Por ejemplo:
  - from time import asctime
  - print (asctime())
- Con *from-import* se indica explícitamente qué función o clase se quiere importar de un módulo

PYTHONPATH

- Cuando hacemos import modulo el interprete busca el módulo en:
  - El directorio desde el que se ejecutó el script de entrada o el directorio actual si el intérprete se ejecuta de forma interactiva
  - La lista de directorios contenidos en la variable de entorno PYTHONPATH.
  - Una lista de directorios dependiente de la instalación configurada en el momento en que se instala Python

PYTHONPATH (2)

- El valor de la variable PYTHONPATH se puede consultar desde Python mediante sys.path
  - import sys
  - print (sys.path)
- *['', '/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages']*

Los módulos también son objetos

- Los módulos son objetos de tipo ***module***.
- Y como todo objeto, tiene atributos y métodos
  - *import sys*
  - *type(sys) → <type 'module'>*
  - *dir(sys) → '__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames', '_getframe', '_mercurial', '_multiarch', 'api_version', 'argv', '……..]*
  - Esta instrucción *dir(sys)*, nos dice qué atributos tiene el objeto
  - En particular, nos interesa uno: *__name___*

Los módulos también son objetos (2)

- Cuando se ejecuta el módulo directamente (o sea, sin ser importado) __name__ tiene como valor “__main__”
- Cuando se importa, el valor de __name__ es el nombre del módulo
- Este comportamiento es útil para hacer que cierto código se ejecute solamente cuando el intérprete llame directamente a un módulo, no cuando el módulo se ejecute producto de un import

```
print("Se muestra siempre")

if __name__ == "__main__":
    print("Se muestra si no es importación")
```

Ejecutando módulos como scripts

- Cuando ejecutamos un módulo de Python con:
  - python mi_modulo.py <argumentos>
- El código en el módulo será ejecutado, tal como si lo hubieses importado (import modulo), pero con __name__ con el valor de "__main__”
- Esto significa que podemos preguntar por el nombre del módulo y distinguir si es un módulo con funciones a importar o un script de ejecución.

```
if __name__ == "__main__":
        import sys
        mi_funcion()
```

Alias

- Es posible también, abreviar los nombres de los paquetes mediante un alias.
- Durante la importación, se asigna la palabra reservada *as* seguida del alias:
  - `import modulo as m`
  - `import paquete.modulo1 as pm`
  - `import paquete.subpaquete.modulo1 as psml`
  - `from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2`
  - `from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1`

Paquetes

- Los paquetes se utilizan para organizar los módulos.
- Los paquetes son tipos especiales de módulos
- Los módulos corresponden a nivel físico con archivos .py, mientras los paquetes representan directorios.
- Para hacer que Python trate a un directorio como un paquete (y pueda ser importado en nuestro código), es necesario crear un archivo **__init__.py** en la carpeta que queremos que sea un paquete.
- Dentro del archivo __init__.py podemos definir variables globales, o bien, dejarlo vacío.

Paquetes (2)

```
paquete
├── __init__.py
├── modulo1.py
├── modulo2.py
└── modulo3.py
```

Los paquetes, a la vez, también pueden contener otros sub-paquetes:

```
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.p
```

Paquetes (3)

- Como los módulos, para importar paquetes también se utiliza import y from y el caracter . para separar paquetes, subpaquetes y módulos

```
# -*- coding: utf-8 *-*


import modulo           # importar un módulo que no pertenece a un paquete
import paquete.modulo1 # importar un módulo que está dentro de un paquete
import paquete.subpaquete.modulo1
```

- Un **namespace** es el nombre que se ha indicado luego de la palabra import, es decir la ruta del módulo, o dicho de otra forma, el paquete en el que se sitúa el módulo

PIP

- Administrador de paquetes estándar para Python.
- Permite instalar y administrar paquetes adicionales.
- Python 3.4
  - A partir de esta versión se incluye en la instalación básica del interprete.
- Versión
  - pip --version

PIP (2)

- Los paquetes se publican en PyPi
  - Existe una gran colección de paquetes extras
- La principal ventaja es la facilidad de su interfaz de línea de comandos (o terminal, o consola), que permite instalar paquetes de softwarePython fácilmente, desde solo una orden:
  - pip install nombre-paquete
- También se puede fácilmente desinstalar
  - *pip uninstall nombre-paquete*
- Proxy:
  - Se puede configurar estas 2 variables de ambiente: de ambiente: HTTP_PROXY y HTTPS_PROXY
  - O bien, se puede ejecutar, desde consola, el comando así: pip install --proxy http://user:password@proxyserver:port Flask

PIP (3)

- Algunos otros comandos útiles de PIP:

```
$ pip install SomePackage              # latest version
$ pip install SomePackage==1.0.4       # specific version
$ pip install 'SomePackage>=1.0.4'     # minimum version
```

- Versión y ubicación: *pip --version*
- Actualizar pip: *python -m pip install --upgrade pip*
- Para un usuario específico: *python -m pip install --upgrade pip -user*
- Actualizar un paquete específico: sudo pip install [package_name] --upgrade
- Ayuda: *pip -help*
- Listar paquetes: *pip list*
- Listar paquetes obsoletos: *pip list -outdated*