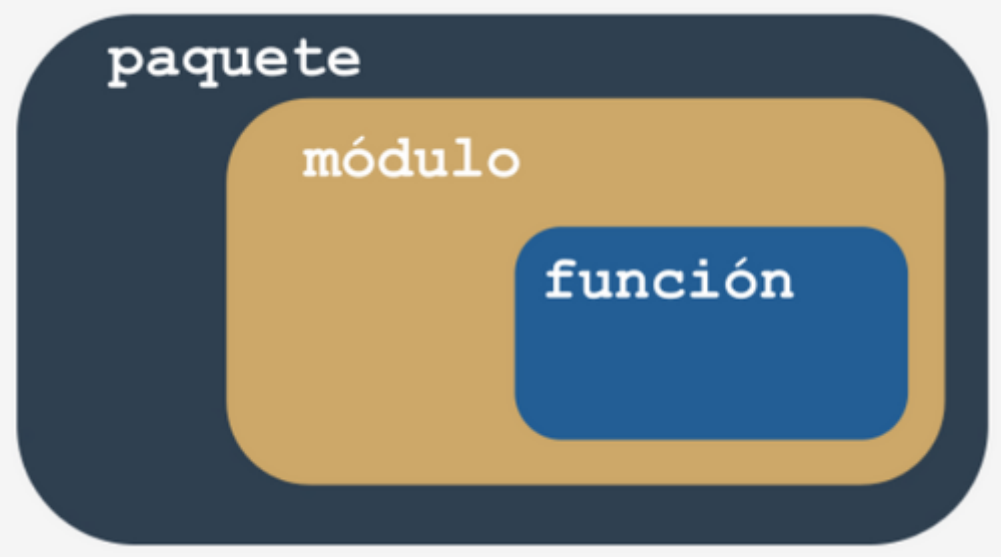
Modulo 1 (intermedio): Paquetes

Escribir tus propios módulos no difiere mucho de escribir scripts comunes.

Existen algunos aspectos específicos que se deben tomar en cuenta, pero definitivamente no es algo complicado. Lo verás pronto.



Resumamos algunos aspectos importantes:

- Un **módulo es un contenedor lleno de funciones** - puedes empaquetar tantas funciones como desees en un módulo y distribuirlo por todo el mundo.
- Por supuesto, no es una buena idea mezclar funciones con diferentes áreas de aplicación dentro de un módulo (al igual que en una biblioteca: nadie espera que los trabajos científicos se incluyan entre los cómics), así que se deben agrupar las funciones cuidadosamente y asignar un nombre claro e intuitivo al módulo que las contiene (por ejemplo, no le des el nombre videojuegos a un módulo que contiene funciones destinadas a particionar y formatear discos duros).
- Crear muchos módulos puede causar desorden: tarde que temprano querrás **agrupar tus módulos** de la misma manera que previamente has agrupado funciones: ¿Existe un contenedor más general que un módulo?
- Sí lo hay, es un **paquete**: en el mundo de los módulos, un paquete juega un papel similar al de una carpeta o directorio en el mundo de los archivos.

Tu primer módulo: paso 1

En esta sección, trabajarás localmente en tu máquina. Comencemos desde cero. Crea un archivo vacío, de la siguiente manera:

module.py

```
```

Se necesitan dos archivos para realizar estos experimentos. Uno de ellos será el módulo en sí. Está vacío ahora. No te preocupes, lo vas a llenar con código pronto.

El archivo lleva por nombre module.py. No muy creativo, pero es simple y claro.

Tu primer módulo: paso 2

El segundo archivo contiene el código que utiliza el nuevo módulo. Su nombre es main.py. Su contenido es muy breve hasta ahora:

Creando el archivo main.py el cual contiene la instrucción import module

main.py

```
import module
```

Nota: ambos archivos deben estar ubicados en la misma carpeta. Te recomendamos crear una carpeta nueva y vacía para ambos archivos. Esto hará que las cosas sean más fáciles.

Inicia el IDLE (o cualquier otro que prefieras) y ejecuta el archivo main.py. ¿Qué es lo que ves?

No deberías ver nada. Esto significa que Python ha importado con éxito el contenido del archivo module.py.

No importa que el módulo esté vacío por ahora. El primer paso ya está hecho, pero antes de dar el siguiente paso, queremos que eches un vistazo a la carpeta en la que se encuentran ambos archivos.

¿Notas algo interesante?

Ha aparecido una nueva subcarpeta, ¿puedes verla? Su nombre es __pycache__. Echa un vistazo adentro. ¿Qué es lo que ves?

Hay un archivo llamado (más o menos) module.cpython-xy.pyc donde x y y son dígitos derivados de tu versión de Python (por ejemplo, serán 3 y 8 si utilizas Python 3.8).

El nombre del archivo es el mismo que el de tu módulo. La parte posterior al primer punto dice qué implementación de Python ha creado el archivo (CPython) y su número de versión. La ultima parte (pyc) viene de las palabras Python y compilado.

Puedes mirar dentro del archivo: el contenido es completamente ilegible para los humanos. Tiene que ser así, ya que el archivo está destinado solo para uso el uso de Python.

Cuando Python importa un módulo por primera vez, traduce el contenido a una forma algo compilada.

El archivo no contiene código en lenguaje máquina: es código semi-compilado interno de Python, listo para ser ejecutado por el intérprete de Python. Como tal archivo no requiere tantas comprobaciones como las de un archivo fuente, la ejecución comienza más rápido y también se ejecuta más rápido.

Gracias a eso, cada importación posterior será más rápida que interpretar el código fuente desde cero.

Python puede verificar si el archivo fuente del módulo ha sido modificado (en este caso, el archivo pyc será reconstruido) o no (cuando el archivo pyc pueda ser ejecutado al instante). Este proceso es completamente automático y transparente, no tiene que ser tomando en cuenta.

Tu primer módulo: paso 3

Ahora hemos puesto algo en el archivo del módulo:

module.py

```
print("Me gusta ser un módulo.")
```

¿Puedes notar alguna diferencia entre un módulo y un script ordinario? No hay ninguna hasta ahora.

Es posible ejecutar este archivo como cualquier otro script. Pruébalo por ti mismo.

¿Qué es lo que pasa? Deberías de ver la siguiente línea dentro de tu consola:

```
Me gusta ser un módulo.
```

Tu primer módulo: paso 4

El archivo main.py file con la instrucción import module

main.py

```
import module
```

Ejecuta el archivo. ¿Qué ves? Con suerte, verás algo como esto:

```
Me gusta ser un módulo.
```

¿Qué significa realmente?

Cuando un módulo es importado, su contenido es ejecutado implícitamente por Python. Le da al módulo la oportunidad de inicializar algunos de sus aspectos internos (por ejemplo, puede asignar a algunas variables valores útiles).

Nota: la inicialización se realiza sólo una vez, cuando se produce la primera importación, por lo que las asignaciones realizadas por el módulo no se repiten innecesariamente.

Imagina el siguiente contexto:

- Existe un módulo llamado mod1.
- Existe un módulo llamado mod2 el cual contiene la instrucción import mod1.
- Hay un archivo principal que contiene las instrucciones import mod1 e import mod2.

A primera vista, se puede pensar que mod1 será importado dos veces - afortunadamente, solo se produce la primera importación. Python recuerda los módulos importados y omite silenciosamente todas las importaciones posteriores.

Tu primer módulo: paso 5

Python puede hacer mucho más que solo importar el módulo. También crea una variable llamada __name__.

Además, cada archivo fuente usa su propia versión separada de la variable, no se comparte entre módulos.

Te mostraremos como usarlo. Modifica el módulo un poco:

module.py

```
print("Me gusta ser un módulo.")
print(__name__)
```

Ahora ejecuta el archivo module.py. Deberías ver las siguientes líneas:

```
Me gusta ser un módulo.
__main__
```

Ahora ejecuta el archivo main.py. ¿Y? ¿Ves lo mismo que nosotros?

```
Me gusta ser un módulo.
module
```

Podemos decir que:

- Cuando se ejecuta un archivo directamente, su variable __name__ se establece a __main__.
- Cuando un archivo se importa como un módulo, su variable __name__ se establece al nombre del archivo (excluyendo a .py).

Tu primer módulo: paso 6

Así es como puedes hacer uso de la variable __main__ para detectar el contexto en el cual se activó tu código:

module.py

```
if __name__ == "__main__":
    print("Yo prefiero ser un módulo")
else:
    print("Me gusta ser un módulo")
```

Sin embargo, existe una forma más inteligente de utilizar la variable. Si escribes un módulo lleno de varias funciones complejas, puedes usarla para colocar una serie de pruebas para verificar si las funciones trabajan correctamente.

Cada vez que modifiques alguna de estas funciones, simplemente puedes ejecutar el módulo para asegurarte de que sus enmiendas no estropeen el código. Estas pruebas se omitirán cuando el código se importe como un módulo.

Tu primer módulo: paso 7

Este módulo contendrá dos funciones simples, y si deseas saber cuantas veces se han invocado las funciones, necesitas un contador inicializado en cero cuando se importe el módulo.

Puedes hacerlo de esta manera:

module.py

```
counter = 0
```

```
if __name__ == "__main__":
    print("Yo prefiero ser un módulo")
else:
    print("Me gusta ser un módulo")
```

Tu primer módulo: paso 8

El introducir tal variable es absolutamente correcto, pero puede causar importantes efectos secundarios que debes tomar en cuenta.

Analiza el archivo modificado main.py:

main.py

```
import module
print(module.counter)
```

Como puedes ver, el archivo principal intenta acceder a la variable de contador del módulo. ¿Es esto legal? Sí lo es. ¿Es utilizable? Claro. ¿Es seguro?

Eso depende: si confías en los usuarios de tu módulo, no hay problema; sin embargo, es posible que no desees que el resto del mundo vea tu **variable personal o privada**.

A diferencia de muchos otros lenguajes de programación, Python no tiene medios para permitirte ocultar tales variables a los ojos de los usuarios del módulo.

Solo puedes informar a tus usuarios que esta es tu variable, que pueden leerla, pero que no deben modificarla bajo ninguna circunstancia.

Esto se hace anteponiendo al nombre de la variable _ (un guión bajo) o __ (dos guiones bajos), pero recuerda, es solo un acuerdo. Los usuarios de tu módulo pueden obedecerlo o no.

Nosotros por supuesto, lo respetaremos. Ahora pongamos dos funciones en el módulo: evaluarán la suma y el producto de los números recopilados en una lista.

Además, agreguemos algunos adornos allí y eliminemos los restos superfluos.

Tu primer módulo: paso 9

Bueno. Escribamos un código nuevo en nuestro archivo module.py. El módulo actualizado está listo aquí:

```
#!/usr/bin/env python3

""" module.py - Un ejemplo de un módulo en Python """

__counter = 0


def suml(the_list):
```

```
    global __counter
    __counter += 1
    the_sum = 0
    for element in the_list:
        the_sum += element
    return the_sum


def prodl(the_list):
    global __counter
    __counter += 1
    prod = 1
    for element in the_list:
        prod *= element
    return prod


if __name__ == "__main__":
    print("Yo prefiero ser un módulo, pero puedo realizar algunas pruebas por ti")
    my_list = [i+1 for i in range(5)]
    print(suml(my_list) == 15)
    print(prodl(my_list) == 120)
```

Algunos elementos necesitan explicación:

- La línea que comienza con #! tiene muchos nombres - puede ser llamada shabang, shebang, hashbang, poundbang o incluso hashpling (no nos preguntes por qué). El nombre en sí no significa nada aquí, su papel es más importante. Desde el punto de vista de Python, es solo un comentario debido a que comienza con #. Para sistemas operativos Unix y similares a Unix (incluido MacOS), dicha línea indica al sistema operativo como ejecutar el contenido del archivo (en otras palabras, que programa debe ejecutarse para interpretar el texto). En algunos entornos (especialmente aquellos conectados con servidores web) la ausencia de esa línea causará problemas.
- Una cadena (quizás una multilínea) colocada antes de las instrucciones de cualquier módulo (incluidas las importaciones) se denomina doc-string, y debe explicar brevemente el propósito y el contenido del módulo.
- Las funciones definidas dentro del módulo (suml() y prodl()) están disponibles para ser importadas.
- Se ha utilizado la variable __name__ para detectar cuando se ejecuta el archivo de forma independiente, y se aprovechó esta oportunidad para realizar algunas pruebas sencillas.

Tu primer módulo: paso 10

Ahora es posible usar el nuevo módulo, esta es una forma de hacerlo:

main.py

```
from module import suml, prodl

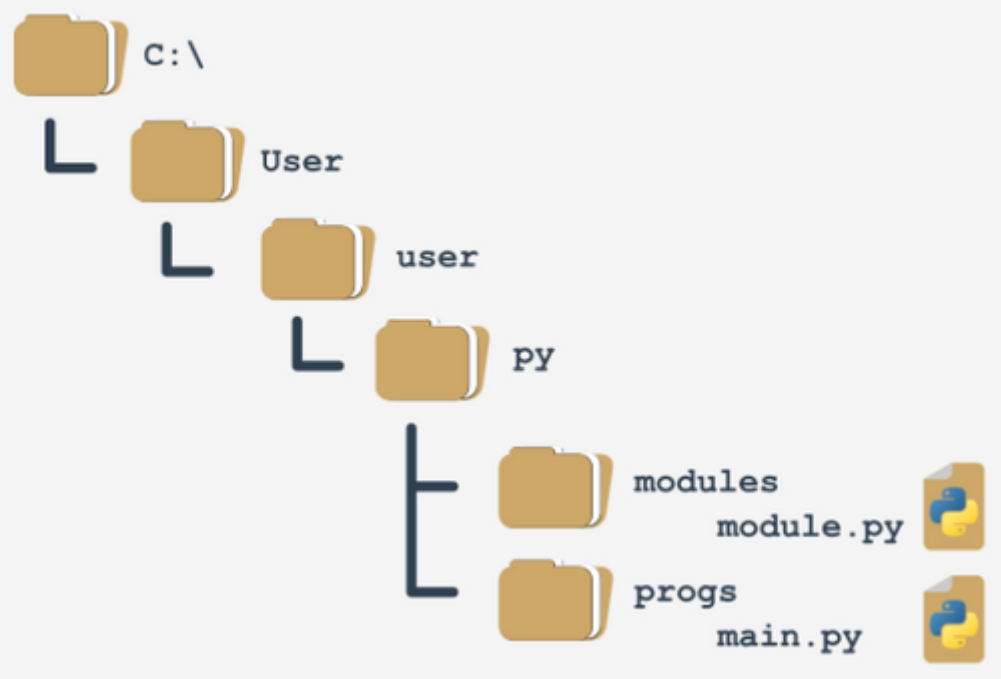
zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(suml(zeroes))
print(prodl(ones))
```

Tu primer módulo: paso 11

Es hora de hacer este ejemplo más complejo: hemos asumido aquí que el archivo Python principal se encuentra en la misma carpeta o directorio que el módulo que se va a importar.

Renunciemos a esta suposición y realicemos el siguiente experimento mental:

- Estamos utilizando el sistema operativo Windows ® (esta suposición es importante, ya que la forma del nombre del archivo depende de ello).
- El script principal de Python se encuentra en C:\Users\user\py\progs y se llama main.py.
- El módulo a importar se encuentra en C:\Users\user\py\modules



¿Cómo lidiar con ello?

Para responder a esta pregunta, tenemos que hablar sobre **como Python busca los módulos**. Hay una variable especial (en realidad una lista) que almacena todas las ubicaciones (carpetas o directorios) que se buscan para encontrar un módulo que ha sido solicitado por la instrucción import.

Python examina estas carpetas en el orden en que aparecen en la lista: si el módulo no se puede encontrar en ninguno de estos directorios, la importación falla.

De lo contrario, se tomará en cuenta la primera carpeta que contenga un módulo con el nombre deseado (si alguna de las carpetas restantes contiene un módulo con ese nombre, se ignorará).

La variable se llama **path** (ruta), y es accesible a través del módulo llamado **sys**. Así es como puedes verificar su valor:

```
import sys

for p in sys.path:
    print(p)
```

Hemos ejecutado el código dentro del directorio C:\User\user y esto es lo que obtenemos:

```
C:\Users\user
```

```
C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip
C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs
C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib
C:\Users\user\AppData\Local\Programs\Python\Python36-32
C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages
```

Nota: la carpeta en la que comienza la ejecución aparece en el **primer elemento de la ruta**.

Ten en cuenta también que: hay un archivo zip listado como uno de los elementos de la ruta, esto no es un error. Python puede tratar los archivos zip como carpetas ordinarias, esto puede ahorrar mucho almacenamiento.

¿Puedes predecir cómo resolver este problema? Puedes resolverlo agregando una carpeta que contenga el módulo a la variable de ruta (la variable path), es completamente modificable.

Tu primer módulo: paso 12

Una de las varias soluciones posibles se ve así:

main.py

```
from sys import path

path.append('..\\modules')

import module

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(module.suml(zeroes))
print(module.prodl(ones))
```

Nota:

- Se ha duplicado la \ dentro del nombre de la carpeta, ¿sabes por qué?
  - Debido a que una diagonal invertida se usa para escapar de otros caracteres, si deseas obtener solo una diagonal invertida, debes escapar.
- Hemos utilizado el nombre relativo de la carpeta: esto funcionará si inicia el archivo main.py directamente desde la carpeta de inicio, y no funcionará si el directorio actual no se ajusta a la ruta relativa; siempre puedes usar una ruta absoluta, como esta:

```
path.append('C:\\Users\\user\\py\\modules')
```

- Hemos usado el método **append()**, la nueva ruta ocupará el último elemento en la lista de rutas; si no te gusta la idea, puedes usar en lugar de ello el método **insert()**.

Tu primer paquete: paso 1

Imagina que en un futuro no muy lejano, tu y tus socios escriben una gran cantidad de funciones en Python.

Tu equipo decide agrupar las funciones en módulos separados, y este es el resultado final:

alpha.py

```
#! /usr/bin/env python3

""" module: alpha """

def funA():
    return "Alpha"

if __name__ == "__main__":
    print("Yo prefiero ser un módulo")
```

Nota: hemos presentado todo el contenido solo para el módulo alpha.py, supongamos que todos los módulos tienen un aspecto similar (contienen una función denominada funX, donde X es la primera letra del nombre del módulo).

Tu primer paquete: paso 2

De repente, alguien se da cuenta de que estos módulos forman su propia jerarquía, por lo que colocarlos a todos en una estructura plana no será una buena idea.

Después de algo de discusión, el equipo llega a la conclusión de que los módulos deben agruparse. Todos los participantes están de acuerdo en que la siguiente estructura de árbol refleja perfectamente las relaciones mutuas entre los módulos:



Repasemos esto de abajo hacia arriba:

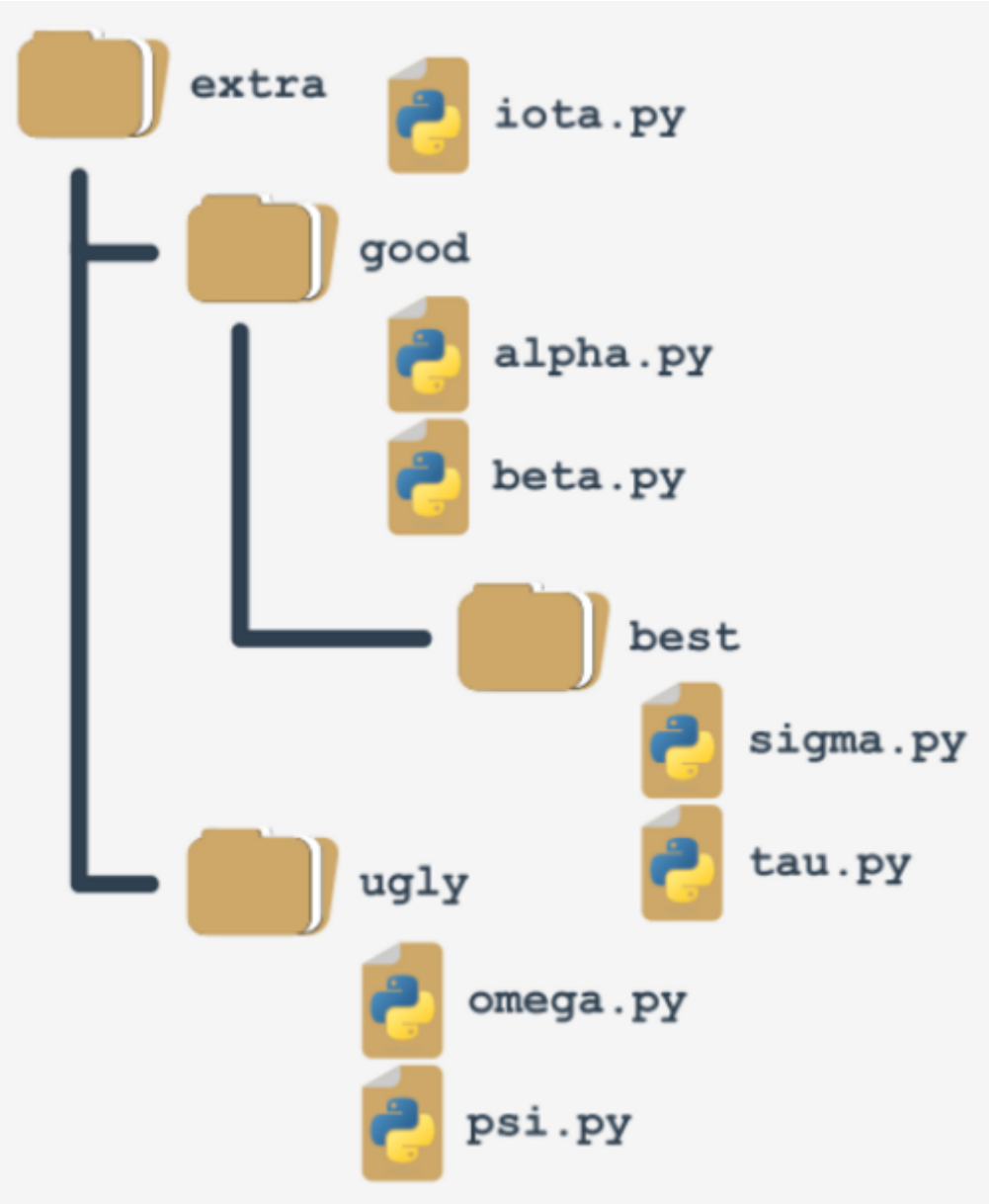
- El grupo ugly contiene dos módulos: psi y omega.
- El grupo best contiene dos módulos: sigma y tau.
- El grupo good contiene dos módulos: (alpha y beta) y un subgrupo (best).
- El grupo extra contiene dos subgrupos: (good y bad) y un módulo (iota).

¿Se ve mal? De ninguna manera: analiza la estructura cuidadosamente. Se parece a algo, ¿no?

Parece la **estructura de un directorio**.

Construyamos un árbol que refleje las dependencias proyectadas entre los módulos.

Así es como se ve el árbol actualmente:



Tal estructura es casi un paquete (en el sentido de Python). Carece del detalle fino para ser funcional y operativo. Lo completaremos en un momento.

Si asumes que **extra** es el nombre de un **paquete recientemente creado** (piensa en el como la **raíz del paquete**), impondrá una regla de nomenclatura que te permitirá nombrar claramente cada entidad del árbol.

Por ejemplo:

La ubicación de una función llamada funT() del paquete tau puede describirse como:

```
extra.good.best.tau.funT()
```

Una función marcada como:

```
extra.ugly.psi.funP()
```

Proviene del módulo psi el cual esta almacenado en el subpaquete ugly del paquete extra.

Tu primer paquete: paso 4

Ahora se deben responder dos preguntas:

- ¿Cómo se transforma este árbol (en realidad, un subárbol) en un paquete real de Python (en otras palabras, ¿cómo convence a Python de que dicho árbol no es solo un montón de archivos basura, sino un conjunto de módulos)?
- ¿Dónde se coloca el subárbol para que Python pueda acceder a él?

La primer pregunta tiene una respuesta sorprendente: **los paquetes, como los módulos, pueden requerir inicialización**.

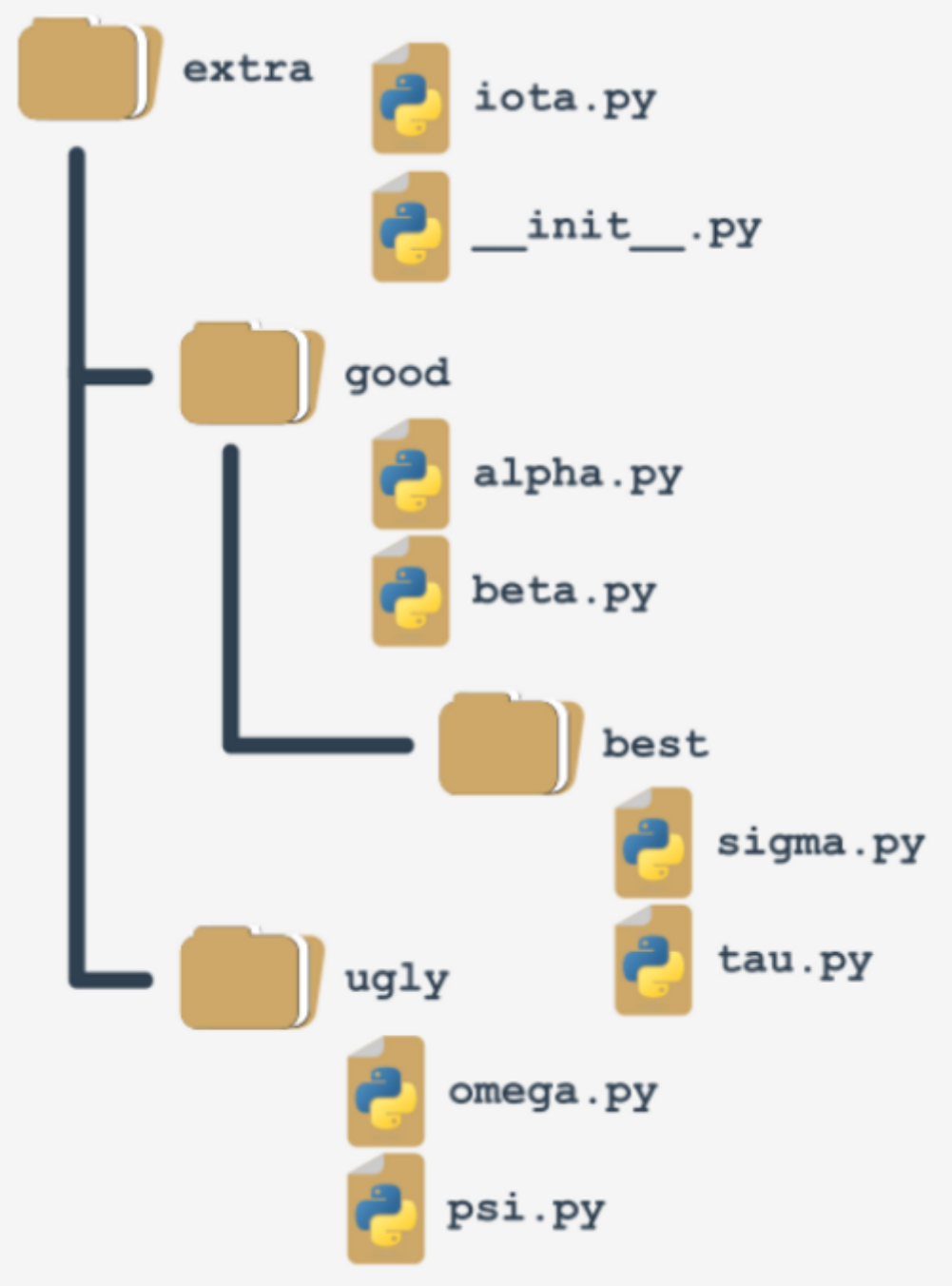
La inicialización de un módulo se realiza mediante un código independiente (que no forma parte de ninguna función) ubicado dentro del archivo del módulo. Como un paquete no es un archivo, esta técnica es inútil para inicializar paquetes.

En su lugar, debes usar un truco diferente: Python espera que haya un archivo con un nombre muy exclusivo dentro de la carpeta del paquete: __init__.py.

El contenido del archivo se ejecuta cuando se **importa** cualquiera de los módulos del paquete. Si no deseas ninguna inicialización especial, puedes dejar el archivo vacío, pero **no debes omitirlo**.

Tu primer paquete: paso 5

Recuerda: la presencia del archivo __init__.py finalmente compone el paquete:



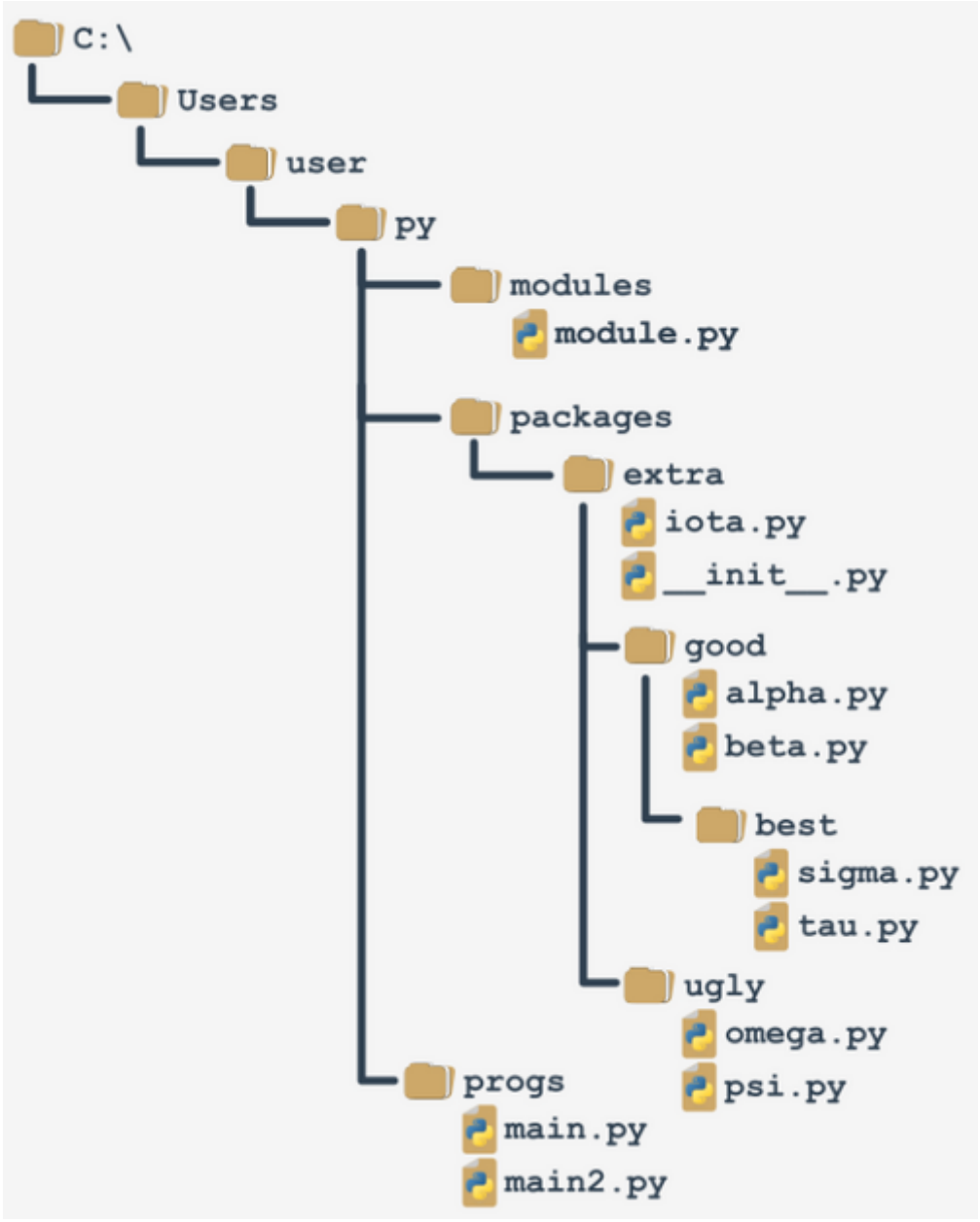
Nota: no solo la carpeta raiz puede contener el archivo __init__.py, también puedes ponerlo dentro de cualquiera de sus subcarpetas (subpaquetes). Puede ser útil si algunos de los subpaquetes requieren tratamiento individual o un tipo especial de inicialización.

Ahora es el momento de responder la segunda pregunta, ¿Dónde se coloca el subárbol para que sea accesible para Python? La respuesta es simple: **donde quiera**. Solo tienes que asegurarte de que Python conozca la ubicación del paquete. Ya sabes como hacer eso.

Estás listo para usar tu primer paquete.

Tu primer paquete: paso 6

Supongamos que el entorno de trabajo se ve de la siguiente manera:



Hemos preparado un archivo zip que contiene todos los archivos de la rama de paquetes. Puedes descargarlo y usarlo para tus propios experimentos, pero recuerda desempaquetarlo en la carpeta presentada en el esquema, de lo contrario, no será accesible para el código.

archivo_zip_modulos_y_paquetes.zip

Continuarás tus experimentos empleado el archivo main2.py.

Tu primer paquete: paso 7

Vamos a acceder a la función funI() del módulo iota del paquete extra. Nos obliga a usar nombres de paquetes calificados (asocia esto al nombramiento de carpetas y subcarpetas).

Así es como se hace:

main2.py

```
from sys import path
path.append('..\\packages')

import extra.iota
print(extra.iota.FunI())
```

Nota:

- Hemos modificado la variable path para que sea accesible a Python. (versión Windows)
- El import no apunta directamente al módulo, pero especifica la ruta completa desde la parte superior del paquete.

El reemplazar import extra.iota con import iota causará un error.

La siguiente variante también es válida:

main2.py

```
from sys import path
path.append('..\\packages')

from extra.iota import FunI
print(FunI())
```

Tu primer paquete: paso 8

Ahora vamos hasta el final del árbol: así es como se obtiene acceso a los módulos sigma y tau.

main2.py

```
from sys import path

path.append('..\\packages')

import extra.good.best.sigma
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())
print(funT())
```

Puedes hacer tu vida más fácil usando un alias:

main2.py

```
from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig
import extra.good.alpha as alp
```

```
print(sig.funS())
print(alp.funA())
```

Tu primer paquete: paso 9

Supongamos que hemos comprimido todo el subdirectorio, comenzando desde la carpeta extra (incluyéndola), y se obtuvo un archivo llamado

extrapack_zip_file.zip

. Después, colocamos el archivo dentro de la carpeta packages.

Ahora podemos usar el archivo zip en un rol de paquetes:

main2.py

```
from sys import path

path.append('..\\packages\\extrapack.zip')

import extra.good.best.sigma as sig
import extra.good.alpha as alp
from extra.iota import funI
from extra.good.beta import funB

print(sig.funS())
print(alp.funA())
print(funI())
print(funB())
```

Si deseas realizar tus propios experimentos con el paquete que hemos creado, puedes descargarlo a continuación. Te alentamos a que lo hagas.

Puntos Clave

1. Mientras que un módulo está diseñado para acoplar algunas entidades relacionadas como funciones, variables o constantes, un paquete es un contenedor que permite el acoplamiento de varios módulos relacionados bajo un mismo nombre. Dicho contenedor se puede distribuir tal cual (como un lote de archivos implementados en un subárbol de directorio) o se puede empaquetar dentro de un archivo zip.

2. Durante la primera importación del módulo, Python traduce su código fuente a un formato semi-compilado almacenado dentro de los archivos pyc y los implementa en el directorio pycache ubicado en el directorio de inicio del módulo.

3. Si deseas decirle al usuario del módulo que una entidad en particular debe tratarse como privada (es decir, no debe usarse explícitamente fuera del módulo), puedes marcar su nombre con el prefijo _ o __. No olvides que esto es solo una recomendación, no una orden.

4. Los nombres shabang, shebang, hasbang, poundbang y hashpling describen el dígrafo escrito como #!, se utiliza para instruir a los sistemas operativos similares a Unix sobre cómo se debe iniciar el archivo fuente de Python. Esta convención no tiene efecto en MS Windows.

5. Si deseas convencer a Python de que debe tomar en cuenta el directorio de un paquete no estándar, su nombre debe insertarse/agregarse en/a la lista de directorios de importación almacenada en la variable path contenida en el módulo sys.

6. Un archivo de Python llamado __init__.py se ejecuta implícitamente cuando un paquete que lo contiene está sujeto a importación y se utiliza para inicializar un paquete y/o sus subpaquetes (si los hay). El archivo puede estar vacío, pero no debe faltar.

From:
https://matebcn.eu/ - **miguel angel torres egea**

Permanent link:
https://matebcn.eu/info:cursos:netacad:python:pe2m1:paquetes

Last update: **23/06/2022 13:18**