SEAS
CAMPUS SEAS

5

UNIDAD DIDÁCTICA 5

Python

5. Módulos y paquetes

ÍNDICE

OBJETIVOS

- Comprender el uso de los módulos y los paquetes en Python.
- Creación de módulos y paquetes.
- Estudiar algunas de las librerías genéricas de uso más frecuente: el módulo *math* y el módulo *random*.
- Aprender a importar componentes de las librerías.
- Estudiar las bases de la programación modular.

INTRODUCCIÓN

En programación, es de vital importancia la agrupación de declaraciones de procedimientos, funciones y constantes para organizar de forma clara y coherente el código de Python ya que permite reutilizar el código en multitud de aplicaciones. Por ejemplo, a nadie se le ocurre tener que escribir una función para calcular el seno de un ángulo. Simplemente habrá una librería de funciones matemáticas entre las que se encuentre dicha función y otras muchas funciones de carácter matemático que pueda necesitar.

En esta unidad se van a ver los módulos y los paquetes de Python y se van a utilizar algunas de las librerías de que dispone el lenguaje. Hay que recalcar que uno de los pilares del éxito de Python es la disponibilidad de un gran conjunto de librerías de uso gratuito para multitud de aplicaciones de ingeniería y ciencias.

En las unidades anteriores hemos visto una colección de útiles que permiten escribir algoritmos claros y correctos para una amplia gama de problemas. Nos referimos a las estructuras de control y a los procedimientos y las funciones. En esta Unidad vamos a ver los módulos, que son una abstracción que da lugar a la denominada programación modular, y los paquetes (*packages*) que agrupan a los componentes en una librería coherente y que se pueden ver como agrupaciones de módulos formando una estructura de orden superior.

5.1. Los módulos en Python

Un módulo es una colección de declaraciones que permiten organizar lógicamente el código Python. Agrupando código relacionado dentro de un módulo resulta más sencillo de entender y de utilizar.

En Python, un módulo se implementa como un fichero con extensión ".py". Puede definir constantes, procedimientos, funciones, variables e incluso código ejecutable. No se requiere una sintaxis especial para que ese fichero sea un módulo.

- Un módulo es una colección de declaraciones que permiten organizar lógicamente el código Python, agrupando código relacionado, para que resulte más sencillo de entender y de usar.
- En Python, un módulo se implementa como un fichero con extensión ".py".
- Para usar un módulo hay que importarlo.

Para que se pueda usar un módulo basta con importarlo. Hay distintas formas de hacerlo, pero en todas ellas se usa la cláusula *import*. Por ejemplo, para importar el módulo *math*, se escribe:

```
>>> import math
```

Los desarrolladores de Python han escrito numerosos módulos, utilizables en multitud de aplicaciones, lo que justifica en parte el éxito del lenguaje. La mayor parte de esos módulos ya están instalados en la versión estándar de Python. Están disponibles en: https://docs.python.org/3/py-modindex.html. Antes de nada, veamos las bases de la descomposición modular.

5.1.1. La descomposición modular

Cuando la tarea de programación a realizar es larga y compleja (por ejemplo, un amplio conjunto de funciones matemáticas, un tratamiento de señales, un procesado estadístico, etc) las herramientas hasta ahora conocidas resultan insuficientes para satisfacer nuestras necesidades de reusabilidad y legibilidad, ya que no es realista tener todo mezclado.

Es fácil observar que lo más razonable es resolver los problemas por separado, reutilizando código que ya se haya escrito, estableciendo los canales de comunicación apropiados entre todos los ámbitos de trabajo para lograr el objetivo final. Los módulos son la herramienta que utilizamos en el proceso de dividir un problema o sus soluciones en partes separadas.

La modularidad nos permite la creación de librerías. Para usar estas librerías, no será necesario conocer la forma en que se han programado. Los usuarios (programadores cliente) de esa librería deben ser capaces de relacionarse con la parte que ellos usan y saber que no necesitarán reescribir el código si se crea una nueva versión de la librería.

Por otra parte, el creador de la librería debe tener libertad para hacer, en versiones posteriores, aquellas modificaciones que considere convenientes con la certeza de que el cliente programador no resultará afectado por tales cambios. Ejemplos de librerías que se van a ver en esta unidad son el módulo *Math* y el módulo *Random*. En la siguiente unidad se verán tres librerías de amplio uso en Ingeniería y Ciencias, las NumPi, Matplotlib y Pandas. Otros módulos de gran interés en Python son:

Módulo	Utilidad	Módulo	Utilidad
re	Expresiones regulares	math	Operaciones matemáticas
difflib	Cálculo en diferencias	random	Números aleatorios
datetime	Fechas y horas	io	Operaciones de bajo nivel sobre flujos de datos
calendar	Calendarios	email	Operaciones sobre correos

Algunos módulos útiles para la programación de redes y de Internet son:

Módulo	Utilidad	Módulo	Utilidad
webbrowser	Gestión de navegadores de Internet	httplib	Programación http
cgi	Herramientas para la programación web en CGI	ftplib	Programación FTP

5.2. Conceptos elementales

Como hemos comentado, un módulo es una colección de declaraciones, ocultas respecto a toda declaración exterior al propio módulo, es decir, definidas en un ámbito de visibilidad cerrado. Los módulos permitirán desarrollar las distintas partes en las que se va a descomponer cualquier programa que tenga cierta entidad.

El módulo da lugar a una metodología de resolución de problemas basada en la descomposición de dichos problemas en soluciones independientes y en la ubicación separada del código que resuelva los distintos problemas. A esta forma de hacer las cosas se la denomina “diseño modular”.

Una gran ventaja que siempre se busca en programación y que esta metodología nos brinda es el poder reutilizar algunas de estas soluciones parciales entre diversos programas, incluso cuando estos se diseñan por distintos programadores.

El fundamento de un buén diseño modular consiste en contemplar nuestros futuros algoritmos como una jerarquía de módulos perfectamente comunicados entre sí, donde cada uno de ellos cuenta con un objetivo diferenciado, como si fueran piezas de una máquina que se pudiesen utilizar para construir otras máquinas.

5.3. Ámbito de los módulos

Al igual que la función, el módulo define un ámbito para los identificadores declarados en él.

Algo a tener en cuenta es que *el ámbito del módulo es estático*. ¿Qué quiere decir? Que el módulo (o la parte importada de él) existe mientras dura la sesión del usuario. Esto significa que las variables declaradas en un módulo también existen -y consumen memoria- a lo largo de toda la duración de la activación del módulo. Estas variables serán por tanto, *variables globales.*

En contraste con estas, cada vez que se ejecuta una función, se abre nuevamente su ámbito – sus variables son *locales* por lo que se definen de nuevo. Cuando termina de ejecutarse la función desaparecen sus variables locales.

5.4. Importaciones

Los objetos declarados en un módulo, tales como variables, constantes, procedimientos y funciones únicamente son visibles en su ámbito que, como se ha comentado, es opaco desde afuera del módulo.

Cuando desde un programa se va a usar código procedente de otros módulos, se debe dejar constancia explícita mediante su importación. Esto se realiza por medio de las listas de importaciones.

Listas de importaciones

La librería de un lenguaje puede contener un gran número de módulos. Evidentemente no resulta práctico que todo el código disponible de ellos sea simultáneamente visible, y por tanto se pueda usar, en todos los módulos.

Una de las cualidades de los módulos es que los programas se pueden escribir por diferentes programadores, que no necesitan conocer las declaraciones hechas en otros módulos, excepto en aquellos que van a usar. La *lista de importaciones* proporciona un mecanismo para evitar los conflictos de nombres. Se emplean las cláusulas **from** e **import**. En realidad siempre se importa lo que sigue a la clásula import. Si le sigue el identificador de un módulo, será un módulo lo que se importe. Si le sigue una lista de elementos tales como constantes, variables, funciones, etc. éstas serán las que se importen; pero en este caso, deberá especificarse el nombre del módulo de procedencia usando la cláusula **from**. Veamos de forma más amplia estas posibilidades:

a. Aquellos elementos de un módulo cuyas declaraciones se desea que sean visibles en un determinado módulo tienen que ser importados, y deben aparecer en la lista de importaciones. Su sintaxis es:

   from <módulo> **import** <lista_importaciones>

   Donde <módulo> es el identificador del módulo del que se desea importar algún elemento, y <lista_importaciones> son los elementos (módulos, subpaquetes, constantes, variables, procedimientos o funciones) a importar separados por coma. Los elementos importados se usarán por su nombre y no se podrán usar estos nombres para otros elementos dentro del programa. Por ejemplo:

```
from math import pi
print(pi)
```

```
3.141592653589793
```

Figura 5.1. Importación de la constante pi del módulo math.

b. Si se desea importar todos los elementos declarados en un módulo se sustituirá la lista_importaciones por un asterisco de la forma:

from <módulo> **import** *

Por ejemplo:

```
from math import *
print(pi)
print(e)
```

3.141592653589793

2.718281828459045

Figura 5.2. Importación del módulo math.

De nuevo se ve que se pueden utilizar las constantes declaradas en el módulo por su nombre, por lo que este nombre no se puede utilizar como identificador para otra cosa al objeto de que no haya conflicto de nombres.

Otra forma de realizar la importación de todos los elementos disponibles en un módulo es:

import <módulo_o_paquete>

Y en este caso, los elementos importados se usarán en forma de un identificador calificado escribiendo el nombre del módulo y un punto seguido del identificador que se use:

<módulo>.<identificador>

De esta forma se subsana el problema de posible duplicidad de los nombres.

Por ejemplo, si se importa el módulo *math* que contiene el número pi, se puede escribir:

```
import math

print(math.pi)
print(math.sin(math.pi/2))
print(math.cos(math.pi/2))
```

3.141592653589793

1.0

6.123233995736766e-17

Figura 5.3. Otra forma de importación del módulo math. Uso del identificador calificado.

Cuando se importa un módulo, se puede cambiar su nombre en la forma:

```
import <modulo> as <nuevo_nombre>
```

Por ejemplo, importamos el módulo math y le llamamos mates escribiendo:

```
import math as mates
```

De forma que en el contexto en que se ha importado el módulo se conoce ahora como mates.

c. A su vez, las funciones contenidas en el módulo también se pueden importar e incluso pueden cambiar de nombre. Por ejemplo:

```
from math import pi, pow as potencia, sin as seno

potencia(2,3)

8
```

Por tanto, se ve que los módulos funcionan como espacios de nombres ya que permiten usar variables con el mismo nombre en el código, salvo cuando se importa en la forma: from <modulo> import *, por lo que se recomienda la forma más general: import <modulo>.

Uso de un módulo como módulo o como programa

Se puede crear un módulo propio que se puede usar como un programa independiente o importarse como un módulo y poder reutilizar sus funciones. Por ejemplo:

Vamos a crear un módulo al que vamos a llamar modA. Su contenido se especifica en la figura 5.4. En él hemos creado tres variables especiales: __dcc__, __autor__ y __versión__ que toman por valores la documentación, el autor del módulo y la versión del mismo.

```
import math
__doc__ = """Soy el módulo modA y trabajo con el módulo
math"""

__autor__ = "Antonio"
__versión__ = "V01"

def raices(a, b, c):
    r = b ** 2 - 4 * a * c
    if r < 0:
        print("Raíces complejas")
        r1 = complex(-b, math.sqrt(-r) / 2)
        r2 = complex(-b, - math.sqrt(-r) / 2)
        if b == 0:
```

```
            print("Raíces imaginarias puras")
            r1 = complex(0, r1)
            r2 = complex(0, r2)
    elif r > 0:
        print("Raíces reales negativas")
        r1 = (-b + math.sqrt®) / 2
        r2 = (-b - math.sqrt®) / 2
    elif r == 0:
            print("Raíces reales negativas dobles")
            r1, r2 = -b / 2, -b / 2
    return r1, r2


if __name__ == "__main__":
    x1 = int(input("Valor de a: "))
    x2 = int(input("Valor de b: "))
    x3 = int(input("Valor de c: "))
    r1, r2 = raices(x1, x2, x3)
    print("( {} , {})".format(r1, r2))
```

Figura 5.4. Módulo modA.

Como se muestra en la figura 5.5, el módulo modA lo podemos ejecutar como programa:

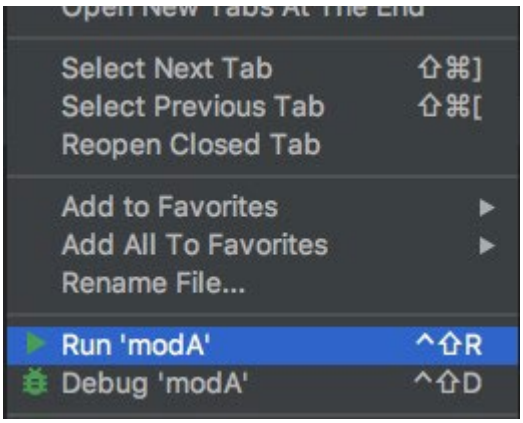

Figura 5.5. Ejecución del módulo modA como programa.

Y si de introducen los valores que se muestran a continuación, el resultado es el que aparece después.

- Valor de a: 1.
- Valor de b: 2.

- Valor de c: 3.

Raíces complejas:

((-2+1.4142135623730951j) , (-2-1.4142135623730951j))

También podemos considerar modA como módulo. Para ello, vamos a crear un nuevo módulo modB que va a importar a modA. Su contenido se muestra en la figura.

```
import modA

docu = modA.__doc__
print(docu)
print(modA.__autor__)
print(modA.__versión__)

x, y = modA.raices(2, 4, 6)
print(x, y)
```

Si se ejecuta modB, el resultado obtenido es.

Soy el módulo modA y trabajo con el módulo math

Antonio

V01

Raíces complejas

(-4+2.8284271247461903j) (-4-2.8284271247461903j)

Por ejemplo, si tras importar el módulo random se ejecuta:

```
print(random.__doc__)
```

aparece en la pantalla:

```
Random variable generators.
    integers
    --------
           uniform within range
    sequences
    ---------
           pick random element
           pick random sample
           pick weighted random sample
           generate random permutation
```

```
    distributions on the real line:
    ------------------------------
           uniform
           triangular
           normal (Gaussian)
           lognormal
           negative exponential
           gamma
           beta
           pareto
           Weibull

    distributions on the circle (angles 0 to 2pi)
    ---------------------------------------------
           circular uniform
           von Mises

General notes on the underlying Mersenne Twister core
generator:
* The period is 2**19937-1.
* It is one of the most extensively tested generators in
existence.
* The random() method is implemented in C, executes in a
single Python step,
  and is, therefore, threadsafe.
```

5.5. Realización de un módulo para generar números aleatorios

Veamos un ejemplo de un módulo que va a incorporar la generación de números pseudoaleatorios.

```
"Generador de números aleatorios"
__autor__="Antonio"
__version__="1.0"

from math import exp

def uniforme ():
    #generación de un número aleatorio comprendido entre 0
y 1
    a,m=16807,2147483647
    q=m//a
    r=m%a
    global z
    g=a*(z%q)-r*(z//q)
    if g>0:
        z=g
    else:
        z=g+m
    return z/m

def exponencial(tao):
    x=uniforme()
    y=exp(-(x-med)/var)
    return y

def normal(med,var):
    x=uniforme()
    y=exp(-(x-med)/var)
    return y

def iniciaSemilla(semilla):
    global z
    z=semilla #necesito dar valor a una variable global desde
fuera, z es local

if __name__=="__main__":
    z=int(input("Valor inicial de semilla: "))
```

Figura 5.6. Módulo para generar números pseudo-aleatorios.

Supongamos que el código anterior se guarda en un módulo de nombre: **aleatorios.py**

En este módulo se pueden hacer los siguientes comentarios:

- En el módulo se utiliza una variable global z. La función *uniforme* opera con dicha variable global y, por lo tanto no necesita parámetros.
- En el módulo se usa una sentencia que asigna un valor inicial a la variable global z.
- El procedimiento *IniciaSemilla* permite inicializar el generador de números aleatorios desde otro módulo, con un valor distinto, al objeto de obtener una secuencia distinta de números.

Desde luego, un generador de números aleatorios es una utilidad que se puede usar en muchas aplicaciones. El módulo que se acaba se escribir, después de almacenado con el nombre "aleatorios.py", queda disponible en forma de módulo objeto, con lo que esta utilidad queda disponible para poderse importar desde otros módulos (clientes de aleatorios). Para ilustrar estas ideas (ya anteriormente expuestas) vamos a considerar el módulo listarAleatorios, que imprime una tabla de 100 números aleatorios.

```
from aleatorios import uniforme,iniciaSemilla
def listar():
    iniciaSemilla(31416)
    max=10
    for i in range(0,max):
        na=uniforme()
        print(na)

listar()
```

Figura 5.7. Uso del módulo anterior.

5.6. Pautas para la agrupación modular

Pasemos ahora a estudiar la forma de descomponer una tarea en módulos de forma que se obtengan programas de mayor calidad.

La idea básica será agrupar variables y funciones, manteniendo una fuerte relación interna. Veamos un ejemplo: se desea un programa que gestione las nóminas de los empleados de una empresa. Para ello se deberá permitir una serie de operaciones como:

- Introducir la nómina de cualquier empleado.
- Consultarla.
- Modificarla.
- Dar de alta un empleado.
- Dar de baja un empleado.
- Imprimir la nómina de un empleado.
- Que al final del año se pueda generar un resumen de las nóminas de todos los empleados.

5.6.1. Agrupación por tareas

Una primera solución al ejemplo anterior consiste en crear módulos separados para implementar las distintas operaciones. De esta forma nos quedaría el esquema mostrado en la figura, donde todos los módulos tienen acceso a los datos globales e interaccionan con el usuario. Este tipo de división del trabajo se denomina "agrupación" por tareas puesto que el criterio principal que se sigue es el de separar las actividades mas o menos independientes. Sin embargo, pese a su aparente naturalidad, este método tiene dos inconvenientes:

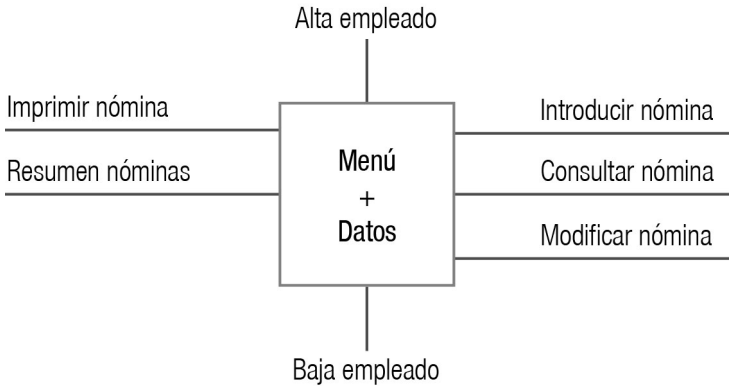

Figura 5.8. Agrupación por tareas.

- Cada vez que se decida efectuar cambios en un *empleado* (por ejemplo, para introducir nuevos datos) o en la implementación de la estructura de datos (típicamente por razones de eficiencia), dado que todos los módulos tienen acceso a esos datos directamente, se tienen que hacer los correspondientes ajustes en todos ellos. Por tanto, no es fácilmente modificable. Lo mismo ocurre con los procedimientos de relación con el usuario.

- No hay protección suficiente para la información: nada impide introducir cambios en los datos de algún empleado dentro del módulo de consulta, lo que no debería suceder.

5.6.2. Agrupación por datos

La manera de evitar los inconvenientes anteriormente mencionados al realizar un diseño modular es la agrupación por datos. En ella se especifican primero los datos que se emplearán y, a continuación, las operaciones asociadas a cada uno.

En nuestro ejemplo, tenemos los tipos *empleado* y *<lista de empleados>*. El primero contiene la información de cada empleado y permite las operaciones: *Dar de alta, Dar de baja, IntroducirNómina, CorregirNómina,* entre otras. La segunda alberga la organización general de los expedientes y operaciones tales como *CrearLista, InsertarEmpleado, BorrarEmpleado, ConsultarNómina,* etc. Por último, la gestión de la secuencia anual de nóminas exige otro módulo.

Ahora ya es posible modificar aspectos internos de cada módulo sin que estos cambios afecten a los demás. También queda garantizado que la única forma de acceder a los datos es mediante los procedimientos exportables construídos a tal efecto.

5.7. La librería estándar de Python

Python viene con una librería muy completa de módulos para todo tipo de aplicaciones tales como acceso a ficheros y directorios, compresión de ficheros, email, xml, csv, aplicaciones matemáticas, procesado de señales, etc.

5.7.1. El módulo math de Python

El módulo *Math* es una librería básica para cálculos numéricos y ofrece las funciones y constantes más frecuentemente utilizadas. Existe otro módulo *CMath* con versiones de esa librería para números complejos.

La definición de dicho módulo contiene:

- Constantes:
  - pi: devuelve una aproximación al número pi.
  - e: devuelve una aproximación al número e.
  - tau: devuelve una aproximación a 2pi.
  - inf: es la representación numérica máxima en coma flotante.
  - nan: es un valor en coma flotante equivalente a “no es un valor numérico”.
- Funciones matemáticas:
  - ceil(x): devuelve el menor entero mayor o igual que x (Redondeo hacia arriba).
  - floor(x): redondeo hacia abajo.
  - copysign(x,y): devuelve un float con el valor absoluto de x, pero con el signo de y.
  - fabs(x): devuelve el valor absoluto de x.
  - factorial(x): devuelve el factorial de x.
  - fmod(x,y): devuelve el resto de dividir x entre y.
  - frexp(x): devuelve la mantisa y exponente de x en forma de un par (m,e), siendo m de tipo float y e entero.
  - fsum(it): devuelve la suma de un iterable.
  - gcd(a,b): devuelve el máximo común divisor de a y b.
  - isclose(a,b,*,rel_tol=1e-09,abs_tol=0.0): devuelve verdad si a y b son próximos y falso en caso contrario.
  - isinfinite(x): devuelve verdad si x no es infinito ni un NaN (*Not* a *number*).

- isinf(x): devuelve verdad si x es infinito positivo o negativo.
- isnan(x): devuelve verdad si x no es un número.
- ldexp(x,i): devuelve x*(2**i)
- modf(x): devuelve las partes fraccional y entera de x.
- remainder(x,y): resto de dividir x entre y.
- trunc(x): devuelve el valor de x truncado a entero.

■ Funciones potenciales y logarítmicas:

- exp(x): devuelve e elevado a x.
- expm1(x): devuelve e elevado a x menos 1.
- log (x [, b]): con un parámetro, devuelve el logaritmo natural (base e) de x. Con dos, devuelve el logaritmo base b de x.
- log1p(x): devuelve el logaritmo natural de 1+x.
- log2(x): devuelve el logaritmo base 2 de x.
- log10(x): devuelve el logaritmo base 10 de x.
- pow(x,y): devuelve x elevado a y.
- sqrt(x): devuelve la raíz cuadrada de x.

■ Funciones trigonométricas.

Los argumentos de todas las funciones trigonométricas e hiperbólicas se deben escribir en radianes y las funciones inversas tanto trigonométricas como hiperbólicas se calculan en radianes (1 radián=180/pi grados).

- acos(x): arco coseno de x.
- asin(x): arco seno de x.
- atan(x): arco tangente de x.
- atan2(x,y): arco cuya tangente es y/x.
- cos(x): coseno de x.
- sin(x): seno de x.
- tan(x): tangente de x.
- hypot(x,y): devuelve la norma euclídea.

- Funciones de conversión angular.
  - degrees(x): convierte el ángulo x de radianes a grados.
  - radian(x): convierte el ángulo x de grados a radianes.
- Funciones hiperbólicas.
  - acosh(x): arco coseno hiperbólico de x
  - asinh(x): arco seno hiperbólico de x
  - atanh(x): arco tangente hiperbólica de x
  - cosh(x): coseno hiperbólico de x
  - sinh(x): sebo hiperbólico de x
  - tanh(x): tangente hiperbólica de x

  A continuación se dan las definiciones de otras funciones trigonométricas e hiperbólicas en términos de las funciones anteriores.

cot(x)=1/math.tan(x)	cosec(x)=1/math.sin(x)
sec(x)=1/math.cos(x)	arcCot=math.pi()/2.0-math.atan(x)
arcCosec(x)=math.asin(1/x)	arcSec(x)=math.acos(1/x)
coth(x)=1/math.tanh(x)	cosech(x)=1/math.sinh(x)
sech(x)=1/math.cosh(x)	arcCoth(x)=math.atanh(1/x)
arcCosech(x)=math.asinh(1/x)	arcSech(x)=math.acosh(1/x)

- Funciones especiales.
  - erf(x): función de error.
  - erfc(x): función de error complementaria.
  - gamma(x): devuelve la función gamma.
  - lgamma(x): devuelve el lgaritmo natural del valor absoluto de la función gamma.

5.7.1.1. Ejemplos

En este apartado se van a trabajar algunos ejemplos con las funciones anteriores del módulo *Math*, al objeto de que su asimilación sea más sencilla. En unidades posteriores se usarán de nuevos estas funciones.

```
import math

x = 2./3.
print('Dos tercios: ', x)
print('ceil: ', math.ceil(x))
print('floor', math.floor(x))
print('truc: ', math.trunc(x))
print('Raiz cuadrada: ', math.sqrt(x))
print('Cuadrado: ', math.pow(x,2))
print('Cubo: ', math.pow(x, 3))

pi = math.pi             # número pi
e = math.e               # número e
x1 = math.log(e)         # logaritmo neperiano de e (vale 1)
x2 = math.log(10, 10)    # logaritmo de 10 base 10 (vale 1)
x3 = math.exp(-0.5)      # función exponencial

x4 = x3 * math.cos(2*math.pi*100*5) # función exponencial
por una senoidal de frecuencia 100 en t=5 segundos

print('PI: ', pi)
print('e: ', e)
print('log(e): ', x1)
print('log base 10 de 10:', x2)
```

```
Dos tercios:  0.6666666666666666
ceil:  1
floor 0
truc:  0
Raiz cuadrada:  0.816496580927726
Cuadrado:  0.4444444444444444
Cubo:  0.2962962962962962
PI:  3.141592653589793
e:  2.718281828459045
log(e):  1.0
log base 10 de 10: 1.0
```

Figura 5.9. Uso de constanes y funciones del módulo math.

En el código que sigue trabajamos con el infinito y con el módulo C*Math* que permite trabajar con números complejos.

```
x5 = math.inf    # trabajamos con el infinito
print(x5)
x6 = x5+100      # infinito más 100 sigue siendo infinito
print(x6)

import cmath             # trabajamos con complejos
x7 = cmath.sqrt(-1)
print(x7)
```

```
inf
inf
1j
```

Figura 5.10. Infinito y uso del módulo cmath.

Si se importan todas las constantes y las funciones del módulo *Math* en la forma:

```
from Math import *
```

No es necesario anteponer *Math* delante de dichas constantes o funciones, como se muestra en el código. Ahora bien, no se podrá dar el mismo nombre a constantes, variables o funciones ya que habría un conflicto de nombres.

```
from math import *
print(pi)
print(e)
print(log(2))
```

```
3.141592653589793
2.718281828459045
0.6931471805599453
```

Figura 5.11. Otra forma de importer el módulo Math.

5.7.2. La función dir.

Esta función devuelve una lista ordenada de cadenas que contienen los identificadores contenidos en el módulo. Por ejemplo, en el caso del módulo *Math*:

```
contenido = dir(math)
print(contenido)
```

Escribe en pantalla:

```
['__doc__', '__file__', '__loader__', '__name__', '__
package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

En donde la variable especial __name__ es una cadena con el nombre del módulo, y __file__ es el nombre del fichero desde el que se ha cargado el módulo.

5.7.3. Módulo random en Python

Este módulo implementa generadores de números pseudo-aleatorios para varias distribuciones. Todas las funciones hacen uso de la función *random()* que es un generador de números pseuso-aleatorios de tipo *float* con distribución uniforme en el intervalo semi-cerrado [0.0, 1.0). Las funciones de este módulo se dividen en funciones para enteros, para secuencias y para reales.

- Para los enteros, hay una función que genera valores con una distribución uniforme dentro de un rango de valores.
- Para las secuencias, hay una que genera valores con distribución uniforme de un elemento, una función para generar una permutación aleatoria de una lista, y una función para el muestreo aleatorio sin sustitución.
- Para números reales, hay funciones para calcular distribuciones:
  - Uniforme:
    - *uniform(a,b)* devuelve un número pseudoaleatorio con distribución uniforme en el intervalo [a, b).
    - *random*, devuelve un número pseudoaleatorio con distribución uniforme en el intervalo [0.0, 1.0)
  - Normales (gaussianas). La función *gauss(mu, sigma)* devuelve una distribución gaussiana de media mu y desviación típica sigma.
  - Distribución de Weibull. La función *weibullvariate(alfa, beta)* devuelve una secuencia con distribución de Weibull, con parámetro de escala alfa y parámetro de forma beta.
  - Otras distribuciones. Están las *lognormales (lognormalvariate(mu, sigma))*, exponenciales negativas *(expovariate(lambda))*, gamma *(gammavariate(alfa*,

beta)) y beta *(betavariate(alfa, beta))*. Para generar distribuciones de ángulos, la distribución de von Mises está disponible *(cunifvariate(medio-arco/2, medio+arco/2))*.

La especificación completa se encuentra en: https://docs.python.org/3/library/random.html

Como ejemplo, vamos a generar 50 valores aleatorios en una lista.

```
import random
random.seed(42)
for i in range(1, 51):
    print(random.random())
```

Figura 5.12. Generación de 50 números pseudo-aleatorios.

5.7.3.1. Ejemplos de uso del módulo random

1. Se va a jugar a los datos que se tiran de dos en dos. Escribir el código para la generación de cuatro tiradas de dados.

```
for i in range(1,5):
    n = random.randint(1,6)
    m = random.randint(1,6)
    print(n, m)
```

```
3 3
2 6
3 6
6 6
```

Figura 5.13. Simulación de cuatro tiradas de dos datos.

Dada una lista, se puede elegir aleatoriamente un elemento de la misma utilizando la función ***choice***. También se puede formar una sublista de forma aleatoria. Por ejemplo:

```
d = [1, 2, 3, 4, 5, 6]
print(random.choice(d))
print(random.choice(d))
print(random.sample(d,4))
```

```
3
6
[6, 5, 2, 3]
```

Figura 5.14. Selección aleatoria de elementos de una lista.

5.8. Paquetes (*packages*)

Si se han creado muchos módulos, se puede perder la información general sobre ellos y sobre su ubicación. Puede haber docenas o cientos de módulos y se pueden clasificar en diferentes categorías. Es similar a la situación en un sistema de archivos: en lugar de tener todos los archivos en un solo directorio, se ubican en diferentes directorios y subdirectorios, organizándose de acuerdo con los temas de los archivos. Pues bien, los módulos relacionados también pueden estar agrupados en paquetes. Por tanto, un paquete es una colección de módulos y/o de paquetes, a veces relacionados entre ellos.

Un paquete es básicamente un directorio con ficheros ".py" y todo paquete contiene un fichero con el nombre: __init__.py. Por tanto, cada directorio que contenga un fichero llamado __init__.py será tratado como un paquete por Python. Cuando se importa un paquete se ejecuta implícitamente la función (pronto la llamaremos método) __init__. Una estructura típica de un paquete puede ser:

```
mi_paquete/
      __init__.py
      modulo1.py
      modulo2.py
      utiles/
            __init__.py
            utiles1.py
            utiles2.py
```

Ejemplo de importaciones pueden ser:

```
import mi_paquete
from mi_paquete import utiles1
```

Por tanto, los paquetes son una forma de estructurar el espacio de nombres de los módulos de Python utilizando el nombre del paquete y a continuación, separados por un punto, el nombre del módulo: <paquete>.<módulo>.

Por ejemplo p.m representa un módulo llamado m en un paquete llamado p. Dos paquetes diferentes, como p1 y p2, pueden tener módulos con el mismo nombre, digamos a, por ejemplo. El módulo a del paquete p1 y el módulo a del paquete p2 pueden ser totalmente diferentes y se designan: p1.a y p2.a. Un paquete se importa como un módulo "normal".

Por ejemplo, vamos a crear un paquete de nombre “pqt”. Para ello, se necesita un directorio que tenga el mismo nombre que el paquete, en este caso, pqt. Este directorio contendrá un fichero de nombre __init__.py que puede estar vacío o puede contener sentencias en Python. Cuando se importa el paquete se ejecutan estas sentencias, por lo que se pueden usar para inicializar un paquete. Ahora ya se pueden poner en este directorio que hemos creado todos los módulos y/o paquetes.

5.8.1. Organización de los paquetes

Al igual que los directorios de los sistemas de ficheros, los paquetes se organizan jerárquicamente y los paquetes pueden contener subpaquetes así como módulos.

En el ejemplo que sigue hay un directorio proyecto, que contiene dos subdirectorios, paquete1 y paquete2. El directorio paquete1 contiene tres ficheros: modulo1.py, modulo2.py y modulo3.py. A su vez, el directorio paquete2 contiene otros cuatro ficheros, los tres módulos: modulo4.py, modulo5.py y modulo6.py, y un fichero de inicialización: __init__.py. También contiene un directorio (subpaquete) que contiene el fichero modulo6.py. En la figura se representa esta estructura en forma gráfica.

```
--- proyecto
      --- paquete1
           --- modulo1.py
           --- modulo2.py
           --- modulo3.py
      --- paquete2
           --- __init__.py
           --- modulo4.py
           --- modulo5.py
           --- subpaquete1
              __init__.py
              --- modulo6.py
```

Vamos a suponer que:

a. El módulo: paquete1/modulo1.py contiene una función: funcion1.

b. El módulo: paquete2/subpaquete1/modulo6.py contiene una función: funcion2.

Ejemplos de importaciones pueden ser:

```
from paquete1 import modulo1

from paquete1.modulo1 import funcion1

from paquete2.subpaquete1.modulo6 import funcion2
```

Cuando se importa paquete2/subpaquete1, se ejecuta implícitamente paquete2/__init__.py y paquete2/subpaquete1/__init__.py.

5.8.2. Creación de paquetes

En primer lugar hay que crear un proyecto nuevo. Como ya sabemos, tal proyecto se crea seleccionando *NewProject* del desplegable *File* del menú principal, tal como se muestra en la figura 5.15.

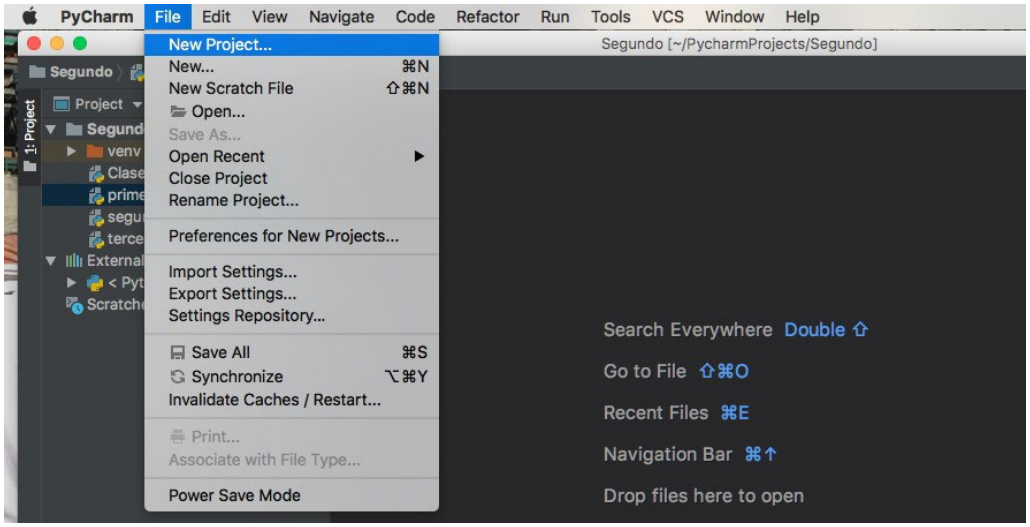

Figura 5.15. Creación de un proyecto nuevo.

En la página web: https://packaging.python.org/tutorials/packaging-projects/ se encuentra información acerca de cómo empaquetar y distribuir un proyecto software.

Tras eso, aparece una ventana para dar nombre al proyecto sustituyendo la palabra *untitled* por el nombre que se quiera dar al proyecto.

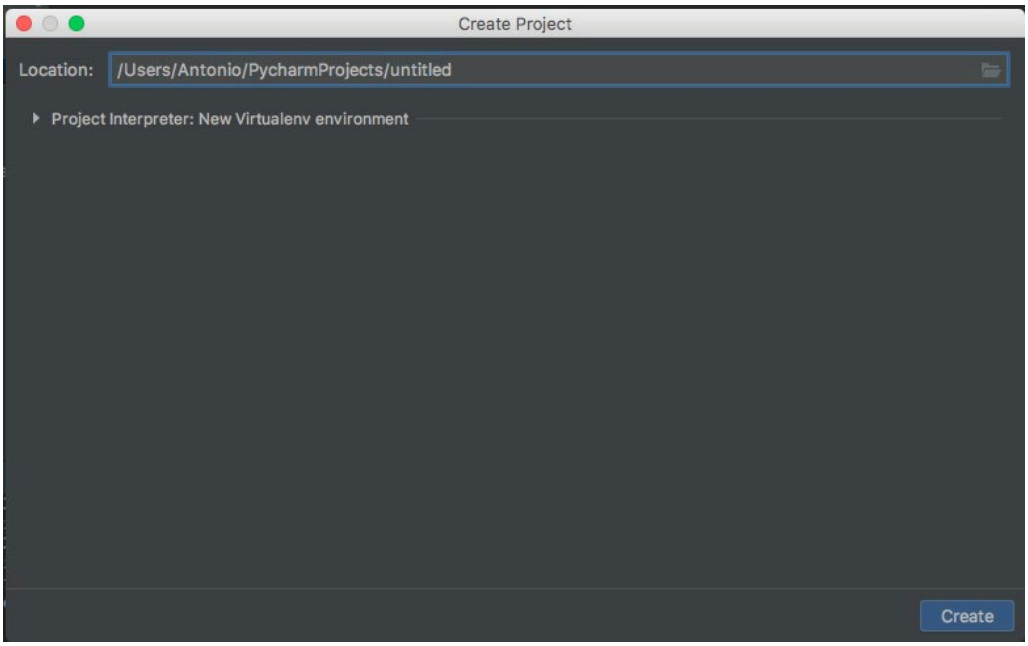

Figura 5.16. Introducción del nombre del proyecto.

Se escribe el nombre, que en nuestro caso, se va a llamar: ProcesadoSeñales.py De esta forma, aparece en forma de carpeta en la parte superior izquierda con su nombre al lado.

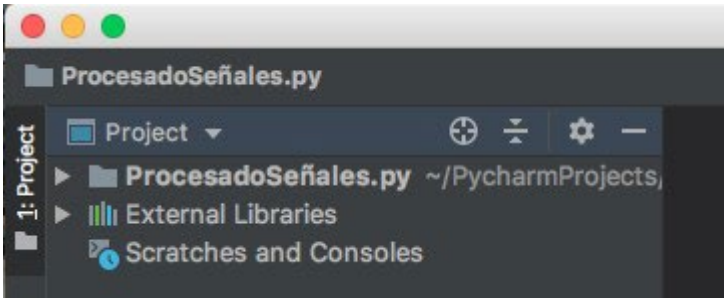

Figura 5.17. Carpeta del proyecto con su nombre.

Como ya sabemos, un paquete es una agrupación de módulos que agrupan un conjunto de módulos relacionados respecto de una determinada aplicación. En realidad un paquete es un directorio que contiene un fichero __init__.py que puede incluso ser vacío y otros ficheros con extensión .py que son los módulos o programas. Cuando se importa un paquete se ejecuta el código contenido en el método __init__(), por lo que se puede usar para inicializar el paquete.

Vamos a crear tres paquetes dentro de este proyecto. Para ello, posicionando el cursor sobre el nombre del proyecto y pulsando el ratón con la tecla derecha aparece un menú desplegable. Nos situamos sobre *New* y vuelve a aparecer otro menú desplegable. Seleccionamos *Python Package*, tal como se muestra en la figura 5.18.

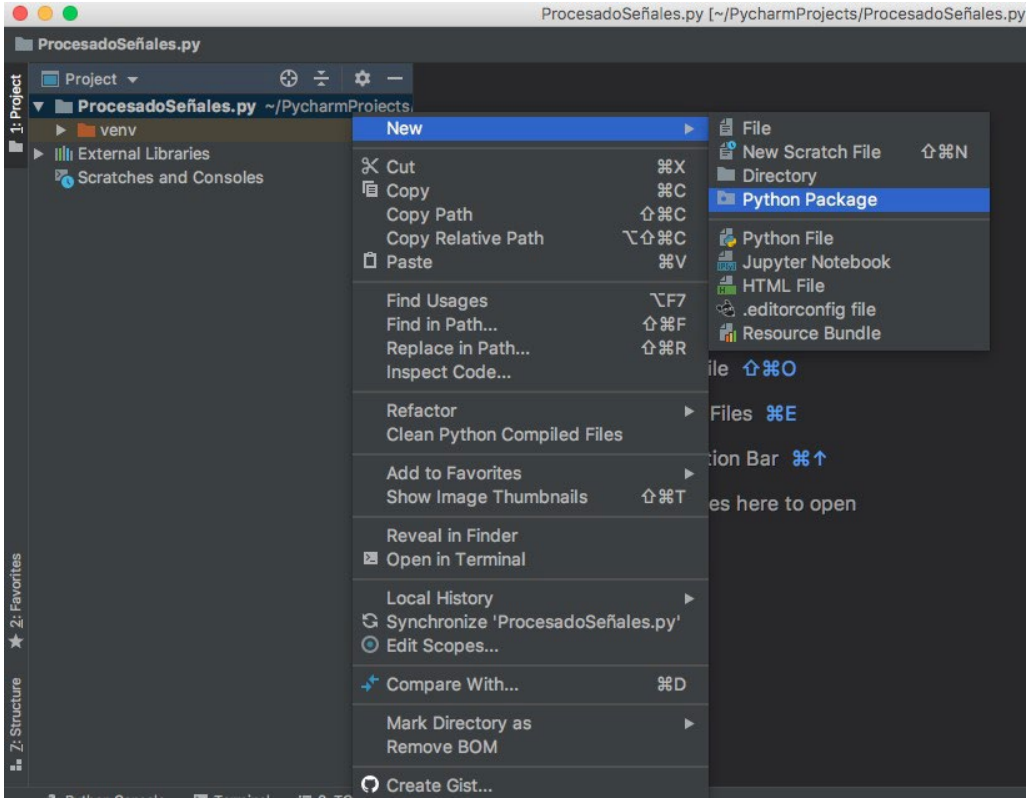

Figura 5.18. Creación del paquete de Python.

Tras eso, aparece una ventana para introducir el nombre del paquete que se quiere crear.

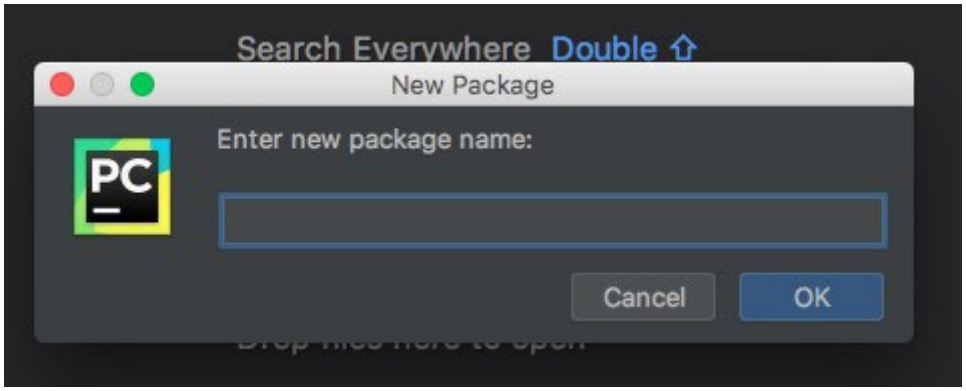

Figura 5.19. Introducción del nombre del paquete.

Y escribimos el nombre del paquete, por ejemplo “Frecuencia”. Tras pulsar OK nos aparecerá debajo del nombre del proyecto, el nombre del paquete al lado de una carpeta agujereada y debajo un fichero de nombre __init__.py que se ha creado automáticamente y que indica que es un paquete de Python. Se puede pulsar sobre los triángulos que aparecen a la izquierda de cada carpeta para presentar o ocultar su contenido.

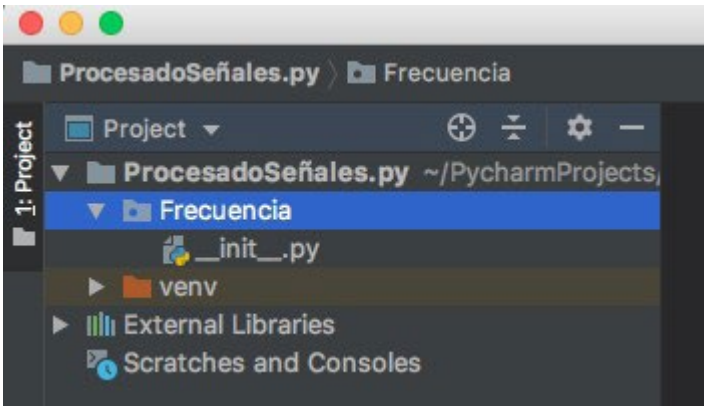

Figura 5.20. Aparece la carpeta del paquete con un fichero __init__.py

De esta forma se pueden crear nuevos paquetes o incluso una jerarquía de paquetes y se pueden añadir módulos a los paquetes para crear una librería. A modo de ejercicio crear dos paquetes adicionales a los que se les llamará: Tiempo y TiempoFrecuencia respectivamente.

Ahora vamos a añadir un módulo dentro del paquete *Frecuencia*. El módulo que vamos a crear es un módulo de ejemplo para ver la forma de utilizar la fft para realizar un procesado de la señal en el dominio de la frecuencia. Para ello, situaremos el cursor del ratón sobre el nombre del paquete. Al pulsar el botón derecho del ratón nos aparece un menú desplegable, y al situarnos con el ratón sobre *New*, nos vuelve a aparecer otro menú desplegable. Lo que queremos es crear un fichero para lo que pulsaremos sobre *File*.

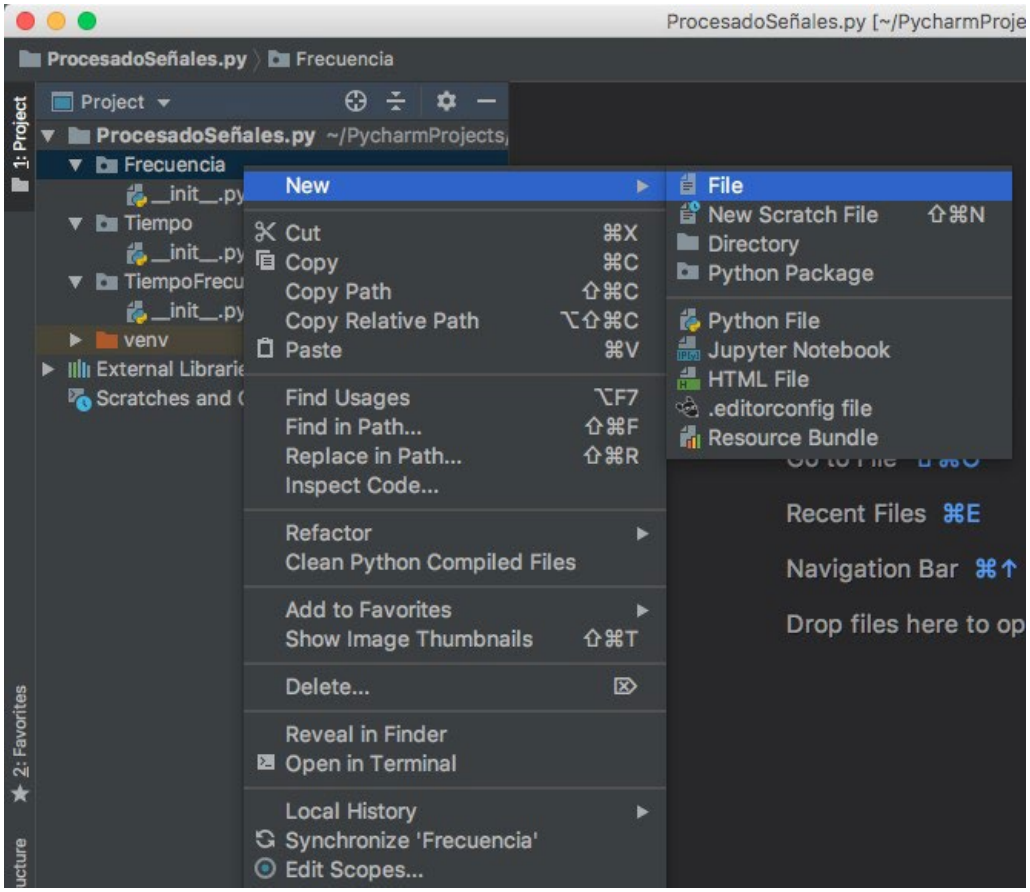

Figura 5.21. Creación de un fichero en el paquete.

Finalmente se pulsa *File* y nos aparece una forma en la que se nos pide el nombre del fichero que queremos crear. Escribamos fft1.py, ya que vamos a usar la función fft para analizar nuestras señales.

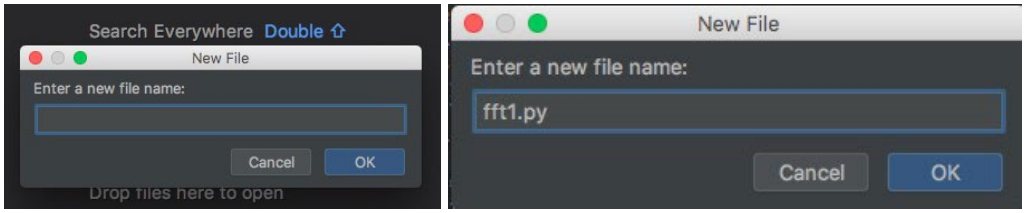

Figura 5.22. Nombre del fichero.

Tras eso, nos aparece un menú para seleccionar el tipo de fichero que queremos. Por defecto viene *Text*, que es el que vamos a seleccionar pulsando OK.

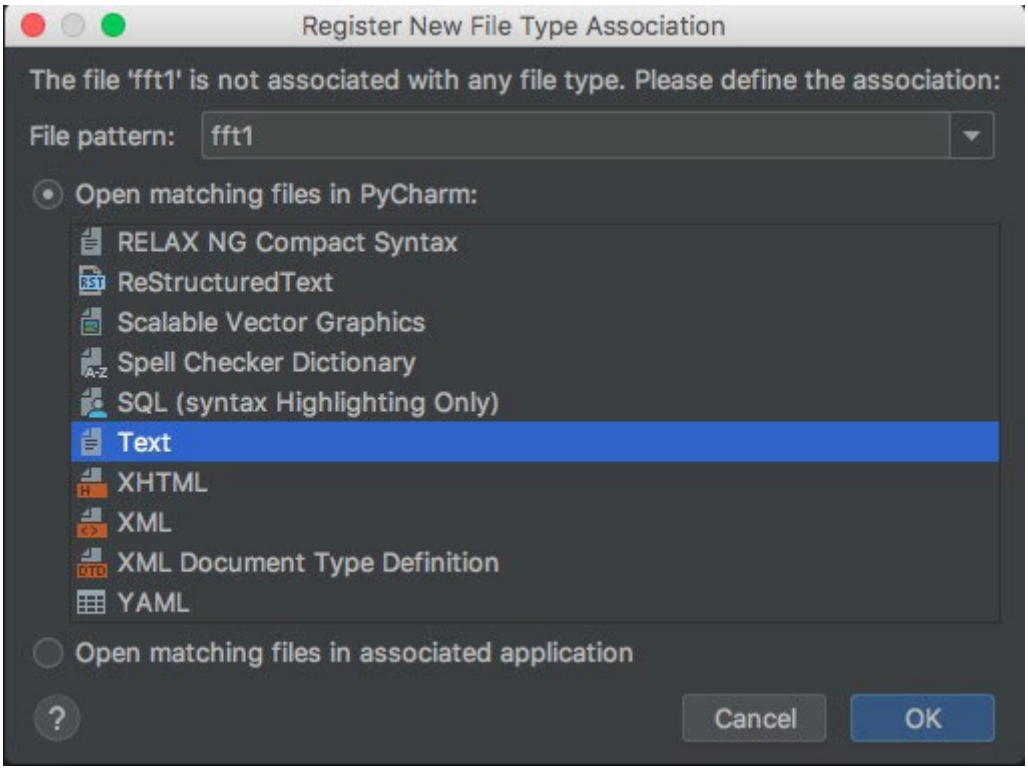

Figura 5.23. Selección del tipo de fichero (de texto).

Ahora nos aparece una pantalla sobre la que se puede escribir el código del módulo correspondiente a la aplicación que vamos a crear para analizar una señal usando la función fft.

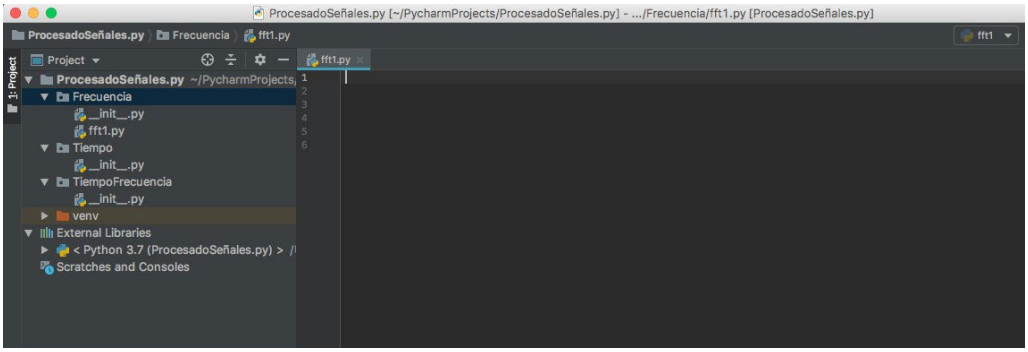

Figura 5.24. Editor del programa.

Para más información ver: https://packaging.python.org/tutorials/packaging-projects/

RESUMEN

- Un módulo es una colección de declaraciones que permiten organizar lógicamente el código Python.
- Un módulo es una colección de declaraciones ocultas respecto a toda declaración exterior al propio módulo, es decir, definidas en un ámbito de visibilidad cerrado.
- Un módulo puede definir constantes, procedimientos, funciones, variables e incluso código ejecutable.
- En Python, un módulo se implementa como un fichero con extensión ".py".
- Para que se pueda usar un módulo basta con importarlo. Hay distintas formas de hacerlo, pero en todas ellas se usa la cláusula *import*.
- Los módulos son la herramienta que utilizamos en el proceso de dividir un problema o sus soluciones en partes separadas.
- El módulo nos permite la creación de librerías.
- El ámbito del módulo es estático ya que lo que se importa existe mientras dura el programa.
- La lista de importaciones proporciona un mecanismo para evitar los conflictos de nombres.
- La cláusula import permite importar un módulo completo. Las claúsulas from <módulo> import <lista_importaciones>, permiten importar elementos declarados en el módulo.
- Se pueden importar todos los elementos declarados en un módulo en la forma: from <módulo> import *. En este caso, se les puede llamar por su nombre, pero ese nombre no se puede usar como identificador en el programa para designar a otros elementos.
- Cuando se importa un módulo en la forma: import <módulo>, a los elementos que contiene el módulo se les llama por su identificador calificado en la forma: <módulo>.<identificador>
- Cuando se importa un módulo o un componente de un módulo se les puede cambiar de nombre (alias) con usando as, en la forma: <nombre> as <alias>.
- Un módulo también se puede usar como un ejecutable. Para esto, antes del código ejecutable se pone: if __name__ == "__main__":
- La variable especial *name* es el nombre del módulo cuando se usa como tal.

- El módulo *Math* es una librería básica para cálculos numéricos y ofrece las funciones y constantes más frecuentemente utilizadas.
- Las funciones trigonométricas trabajan con radianes.
- El módulo Random implementa generadores de números pseudo-aleatorios para varias distribuciones.
- Todas las funciones hacen uso de la función *random()* que es un generador de números pseuso-aleatorios de tipo *float* con distribución uniforme en el intervalo semi-cerrado [0.0, 1.0).
- Las funciones del módulo *random* se dividen en funciones para enteros, para secuencias y para reales.
- Los módulos relacionados también pueden estar agrupados en paquetes. Por tanto, un paquete es una colección de módulos y/o de paquetes, a veces relacionados entre ellos.
- Un paquete es básicamente un directorio con ficheros ".py" y todo paquete contiene un fichero con el nombre: __init__.py.
- Al igual que los directorios de los sistemas de ficheros, los paquetes se organizan jerárquicamente y los paquetes pueden contener subpaquetes así como módulos.