Modulo 1 (intermedio): Módulos

El código de computadora tiene una tendencia a crecer. Podemos decir que el código que no crece probablemente sea completamente inutilizable o esté abandonado. Un código real, deseado y ampliamente utilizado se desarrolla continuamente, ya que tanto las demandas de los usuarios como sus expectativas se desarrollan de manera diferente.

Un código que no puede responder a las necesidades de los usuarios se olvidará rápidamente y se reemplazará instantáneamente con un código nuevo, mejor y más flexible. Se debe estar preparado para esto, y nunca pienses que tus programas están terminados por completo. La finalización es un estado de transición y generalmente pasa rápidamente, después del primer informe de error. Python en sí es un buen ejemplo de cómo actúa esta regla.

El código creciente es, de hecho, un problema creciente. Un código más grande siempre significa un mantenimiento más difícil. La búsqueda de errores siempre es más fácil cuando el código es más pequeño (al igual que encontrar una rotura mecánica es más simple cuando la maquinaria es más simple y pequeña).

Además, cuando se espera que el código que se está creando sea realmente grande (puedes usar el número total de líneas de código como una medida útil, pero no muy precisa, del tamaño del código) entonces, se deseará, o más bien, habrá la necesidad de dividirlo en muchas partes, implementado en paralelo por unos cuantos, una docena, varias docenas o incluso varios cientos de desarrolladores.

Por supuesto, esto no se puede hacer usando un archivo fuente grande, el cual esta siendo editado por todos los programadores al mismo tiempo. Esto seguramente conducirá a un desastre.

Si se desea que dicho proyecto de software se complete con éxito, se deben tener los medios que permitan:

- Dividir todas las tareas entre los desarrolladores.
- Después, unir todas las partes creadas en un todo funcional.

Por ejemplo, un determinado proyecto se puede dividir en dos partes principales:

- La interfaz de usuario (la parte que se comunica con el usuario mediante widgets y una pantalla gráfica).
- La lógica (la parte que procesa los datos y produce resultados).

Cada una de estas partes se puede (muy probablemente) dividir en otras más pequeñas, y así sucesivamente. Tal proceso a menudo se denomina descomposición.

Por ejemplo, si te pidieran organizar una boda, no harías todo tu mismo: encontrarías una serie de profesionales y dividirías la tarea entre todos.

¿Cómo se divide una pieza de software en partes separadas pero cooperantes? Esta es la pregunta. Los módulos son la respuesta.

¿Cómo hacer uso de un módulo?

Entonces, ¿qué es un módulo? El Tutorial de Python lo define como un archivo que contiene definiciones y sentencias de Python, que se pueden importar más tarde y utilizar cuando sea necesario.

El manejo de los módulos consta de dos cuestiones diferentes:

- El primero (probablemente el más común) ocurre cuando se desea utilizar un módulo ya existente, escrito por otra persona o creado por el programador mismo en algún proyecto complejo: en este caso, se considera al programador como el usuario del módulo.

- El segundo ocurre cuando se desea crear un nuevo módulo, ya sea para uso propio o para facilitar la vida de otros programadores: aquí tu eres el proveedor del módulo.

Discutamos ambos por separado.

En primer lugar, un módulo se identifica por su **nombre**. Si se desea utilizar cualquier módulo, se necesita saber su nombre. Se entrega una cantidad (bastante grande) de módulos junto con Python. Se puede pensar en ellos como una especie de «equipamiento adicional de Python».

Todos estos módulos, junto con las funciones integradas, forman la **Biblioteca Estándar de Python** - un tipo especial de biblioteca donde los módulos desempeñan el papel de libros (incluso podemos decir que las carpetas desempeñan el papel de estanterías). Si deseas ver la lista completa de todos los «volúmenes» recopilados en esa biblioteca, se puede encontrar aquí: https://docs.python.org/3/library/index.html.

Cada módulo consta de entidades (como un libro consta de capítulos). Estas entidades pueden ser funciones, variables, constantes, clases y objetos. Si se sabe como acceder a un módulo en particular, se puede utilizar cualquiera de las entidades que almacena.

Comencemos la discusión con uno de los módulos más utilizados, el que lleva por nombre **math**. Su nombre habla por sí mismo: el módulo contiene una rica colección de entidades (no solo funciones) que permiten a un programador implementar efectivamente cálculos que exigen el uso de funciones matemáticas, como **sen()** o **log()**.

Importando un módulo

Para que un módulo sea utilizable, hay que importarlo (piensa en ello como sacar un libro del estante). La importación de un módulo se realiza mediante una instrucción llamada **import**. Nota: import es también una palabra clave reservada (con todas sus implicaciones).

Supongamos que deseas utilizar dos entidades proporcionadas por el módulo **math**:

- Un símbolo (constante) que representa un valor preciso (tan preciso como sea posible usando aritmética de punto flotante doble) de π (aunque usar una letra griega para nombrar una variable es totalmente posible en Python, el símbolo se llama pi: es una solución más conveniente, especialmente para esa parte del mundo que ni tiene ni va a usar un Teclado Griego).
- Una función llamada sin() (el equivalente informático de la función matemática seno).

Ambas entidades están disponibles a través del módulo math, pero la forma en que se pueden usar depende en gran medida de como se haya realizado la importación.

La forma más sencilla de importar un módulo en particular es usar la instrucción para importar de la siguiente manera:

```
import math
```

La cláusula contiene:

- La palabra reservada import.
- El **nombre del módulo** que se va a importar.

La instrucción puede colocarse en cualquier parte del código, pero debe colocarse **antes del primer uso de cualquiera de las entidades del módulo**.

Si se desea (o se tiene que) importar más de un módulo, se puede hacer repitiendo la cláusula **import**, o listando los módulos despues de la palabra reservada import, por ejemplo:

```
import math
import sys
```

o enumerando los módulos después de la palabra clave reservada import, como aquí:

```
import math, sys
```

La instrucción importa dos módulos, primero uno llamado math y luego un segundo llamado sys.

La lista de módulos puede ser arbitrariamente larga.

Para continuar, debes familiarizarte con un término importante: **namespace**. No te preocupes, no entraremos en detalles: esta explicación será lo más breve posible.

Un **namespace** es un espacio (entendido en un contexto no físico) en el que existen algunos nombres y los nombres no entran en conflicto entre sí (es decir, no hay dos objetos diferentes con el mismo nombre). Podemos decir que cada grupo social es un namespace - el grupo tiende a nombrar a cada uno de sus miembros de una manera única (por ejemplo, los padres no darían a sus hijos el mismo nombre).



Esta singularidad se puede lograr de muchas maneras, por ejemplo, mediante el uso de apodos junto con los nombres (funcionará dentro de un grupo pequeño como una clase en una escuela) o asignando identificadores especiales a todos los miembros del grupo (el número de Seguro Social de EE. UU. es un buen ejemplo de tal práctica).

Dentro de un determinado namespace, cada nombre debe permanecer único. Esto puede significar que algunos nombres pueden desaparecer cuando cualquier otra entidad de un nombre ya conocido ingresa al namespace. Mostraremos como funciona y como controlarlo, pero primero, volvamos a las importaciones.

Si el módulo de un nombre especificado **existe y es accesible** (un módulo es de hecho un **archivo fuente de Python**), Python importa su contenido, **se hacen conocidos todos los nombres definidos en el módulo**, pero no ingresan al namespace del código.

Esto significa que puedes tener tus propias entidades llamadas **sin** o **pi** y no serán afectadas en alguna manera por la importación.

En este punto, es posible que te estes preguntando como acceder al **pi** el cual viene del módulo **math**.

Para hacer esto, se debe de mandar llamar el pi con el nombre del módulo original.

Observa el fragmento a continuación, esta es la forma en que se habilitan los nombres de pi y sin con el nombre de su módulo de origen:

```
math.pi
math.sin
```

Es sencillo, se pone:

- El **nombre del módulo** (math).
- Un **punto**.
- El **nombre de la entidad** (pi).

Tal forma indica claramente el namespace en el que existe el nombre.

Nota: el uso de esto es **obligatorio** si un módulo ha sido importado con la instrucción import. No importa si alguno de los nombres del código y del namespace del módulo están en conflicto o no.

Este primer ejemplo no será muy avanzado: solo se desea imprimir el valor de **sin(½π)**.

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

El código genera el valor esperado: **1.0**.

Nota: el eliminar cualquiera de las dos indicaciones del nombre del módulo hará que el código sea erróneo. No hay otra forma de entrar al namespace de math si se hizo lo siguiente:

```
import math
```

Ahora te mostraremos cómo pueden dos namespaces (el tuyo y el del módulo) pueden coexistir.

```
import math

def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None

pi = 3.14

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

Hemos definido el nuestro propio pi y sin aquí.

Ejecuta el programa. El código debe producir la siguiente salida:

```
0.99999999
1.0
```

Como puedes ver, las entidades no se afectan entre sí.

En el segundo método, la sintaxis del import señala con precisión que entidad (o entidades) del módulo son aceptables en el código:

```
from math import pi
```

La instrucción consta de los siguientes elementos:

- La palabra clave reservada **from**.
- El nombre del módulo a ser (selectivamente) importado.
- La palabra clave reservada **import**.
- El nombre o lista de nombres de la entidad o entidades las cuales estan siendo importadas al namespace.

La instrucción tiene este efecto:

- Las entidades listadas son las únicas que son importadas del módulo indicado.
- Los nombres de las entidades importadas pueden ser accedidas dentro del código sin especificar el

nombre del módulo de origen.

Nota: no se importan otras entidades, únicamente las especificadas. Además, no se pueden importar entidades adicionales utilizando una línea como esta:

```
print(math.e)
```

Esto ocasionará un error, (e es la constante de Euler: 2.71828...).

Reescribamos el código anterior para incorporar esta nueva técnica.

```
from math import sin, pi

print(sin(pi/2))
```

El resultado debe de ser el mismo que el anterior, se han empleado las mismas entidades: 1.0. Copia y pega el código en el editor, y ejecuta el programa.

```
from math import sin, pi

print(sin(pi / 2))

pi = 3.14


def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None


print(sin(pi / 2))
```

- La línea 01: lleva a cabo la importación selectiva.
- La línea 03: hace uso de las entidades importadas y obtiene el resultado esperado (1.0).
- La líneas 05 a la 12: redefinen el significado de pi y sin - en efecto, reemplazan las definiciones originales (importadas) dentro del namespace del código.
- La línea 15: retorna 0.99999999, lo cual confirma nuestras conclusiones.

Hagamos otra prueba. Observa el código a continuación:

```
pi = 3.14


def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None
```

```
print(sin(pi / 2))

from math import sin, pi

print(sin(pi / 2))
```

Aquí, se ha invertido la secuencia de las operaciones del código:

- Las líneas del 01 al 08: definen nuestro propio pi y sin.
- La línea 11: hace uso de ellas (0.99999999 aparece en pantalla).
- La línea 13: lleva a cabo la importación - los símbolos importados reemplazan sus definiciones anteriores dentro del namespace.
- La línea 15: retorna 1.0 como resultado.

Importando un módulo: *

En el tercer método, la sintaxis del import es una forma más agresiva que la presentada anteriormente:

```
from module import *
```

Como puedes ver, el nombre de una entidad (o la lista de nombres de entidades) se reemplaza con un solo asterisco (*).

Tal instrucción **importa todas las entidades del módulo indicado**.

¿Es conveniente? Sí, lo es, ya que libera del deber de enumerar todos los nombres que se necesiten.

¿Es inseguro? Sí, a menos que conozcas todos los nombres proporcionados por el módulo, **es posible que no puedas evitar conflictos de nombres**. Trata esto como una solución temporal e intenta no usarlo en un código regular.

Importando un módulo: la palabra clave reservada as

Si se importa un módulo y no se esta conforme con el nombre del módulo en particular (por ejemplo, si es el mismo que el de una de sus entidades ya definidas) puede dársele otro nombre: esto se llama **aliasing** o renombrado.

Aliasing (renombrado) hace que el módulo se identifique con un nombre diferente al original. Esto también puede acortar los nombres originales.

La creación de un alias se realiza junto con la importación del módulo, y exige la siguiente forma de la instrucción import:

```
import module as alias
```

El «module» identifica el nombre del módulo original mientras que el «alias» es el nombre que se desea usar en lugar del original.

Nota: **as** es una palabra clave reservada.

Si necesitas cambiar la palabra math, puedes introducir tu propio nombre, como en el ejemplo:

```
import math as m
```

```
print(m.sin(m.pi/2))
```

Nota: después de la ejecución exitosa de una importación con alias, el nombre original del módulo se vuelve inaccesible y no debe ser utilizado.

A su vez, cuando usa la variante from module import name y se necesita cambiar el nombre de la entidad, se crea un alias para la entidad. Esto hará que el nombre sea reemplazado por el alias que se elija.

Así es como se puede hacer:

```
from module import name as alias
```

Como anteriormente, el nombre original (sin alias) se vuelve inaccesible.

La frase **name as alias** puede repetirse: puedes emplear comas para separar las frases, como a continuación:

```
from module import n as a, m as b, o as c
```

El ejemplo puede parecer un poco extraño, pero funciona:

```
<codepython>
from math import pi as PI, sin as sine
```

print(sine(PI/2)) </code>

```
Ahora etás familiarizado con los conceptos básicos del uso de módulos. Permítenos
mostrarte algunos módulos y algunas de sus entidades útiles.

== Punt
```

os Claves 1. Si deseas importar un módulo como un todo, puedes hacerlo usando la sentencia import nombre_del_módulo. Puedes importar más de un módulo a la vez utilizando una lista separada por comas. Por ejemplo:

```
<code python>
import mod1import mod2, mod3, mod4
```

Aunque la ú</code>ltima forma no se recomienda por razones estilísticas, y es mejor y más bonito expresar la misma intención de una f rma más detallada y explícita, como por ejemplo:

```
import mod2
import mod3
import mod4
```

2. Si un módulo se importa de la manera anterior y desea acceder a cualquiera de sus entidades, debes anteponer el nombre de la entidad empleando la **notación con punto**. Por ejemplo:

```
import my_module

result = my_module.my_function(my_module.my_data)
```

El fragmento de código utiliza dos entidades que provienen del módulo my_module: una función llamada my_function() y una variable con el nombre my_data. Ambos nombres **deben tener el prefijo my_module**. Ninguno de los nombres de entidad importados entra en conflicto con los nombres idénticos existentes en el namespace de tu código.

3. Se te permite no solo importar un módulo como un todo, sino también importar solo entidades individuales de él. En este caso, las entidades importadas no deben especificar el prefijo cuando son empleadas. Por ejemplo:

```
from module import my_function, my_data

result = my_function(my_data)
```

La forma anterior, a pesar de su atractivo, no se recomienda debido al peligro de causar conflictos con los nombres derivados de la importación del namespace del código.

4. La forma más general de la sentencia anterior te permite importar **todas las entidades** ofrecidas por un módulo:

```
from my_module import *

result = my_function(my_data)
```

Nota: la variante de esta importación no se recomienda debido a las mismas razones que antes (la amenaza de un conflicto de nombres es aún más peligrosa aquí).

5. Puede cambiar el nombre de la entidad importada «sobre la marcha» utilizando la frase as del import. Por ejemplo:

```
from module import my_function as fun, my_data as dat

result = fun(dat)
```

Módulos útiles

Trabajando con módulos estándar

Antes de comenzar a revisar algunos módulos estándar de Python, veamos la función **dir()**. No tiene nada que ver con el comando dir de las terminales de Windows o Unix. El comando dir() no muestra el contenido de un directorio o carpeta de disco, pero no se puede negar que hace algo similar: puede revelar todos los nombres proporcionados a través de un módulo en particular.

Existe una condición: el módulo debe haberse importado previamente como un todo (es decir, utilizar la instrucción import module - from module no es suficiente).

La función devuelve una **lista ordenada alfabéticamente** la cual contiene todos los nombres de las entidades disponibles en el módulo:

```
dir(module)
```

Nota: Si el nombre del módulo tiene un alias, debes usar el alias, no el nombre original.

Usar la función dentro de un script normal no tiene mucho sentido, pero aún así, es posible.

Por ejemplo, se puede ejecutar el siguiente código para imprimir los nombres de todas las entidades dentro del módulo math:

```
import math

```

```
for name in dir(math):
    print(name, end="\t")
```

El código del ejemplo debería producir el siguiente resultado:

```
__doc__ __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    hypot    isfinite    isinf    isnan    ldexp    lgamma    log    log10
log1p    log2    modf    pi    pow    radians    sin    sinh    sqrt    tan    tanh
trunc
```

¿Has notado los nombres extraños que comienzan con __ al inicio de la lista? Se hablará más sobre ellos cuando hablemos sobre los problemas relacionados con la escritura de módulos propios.

Algunos de los nombres pueden traer recuerdos de las lecciones de matemáticas, y probablemente no tendrás ningún problema en adivinar su significado.

El emplear la función **dir()** dentro de un código puede no parecer muy útil; por lo general, se desea conocer el contenido de un módulo en particular antes de escribir y ejecutar el código.

Afortunadamente, se puede ejecutar la función directamente en la consola de Python (IDLE), sin necesidad de escribir y ejecutar un script por separado.

Funciones seleccionadas del módulo math

Comencemos con una vista previa de algunas de las funciones proporcionadas por el módulo math.

Se han elegido algunas arbitrariamente, pero esto no significa que las funciones no mencionadas aquí sean menos significativas. Tómate el tiempo para revisar las demás por ti mismo: no tenemos el espacio ni el tiempo para hablar de todas a detalle.

El primer grupo de funciones de módulo math están relacionadas con **trigonometría**:

- sin(x) → el seno de x.
- cos(x) → el coseno de x.
- tan(x) → la tangente de x.

Todas estas funciones toman un argumento (una medida de ángulo expresada en radianes) y devuelven el resultado apropiado (ten cuidado con tan() - no todos los argumentos son aceptados).

Por supuesto, también están sus versiones inversas:

- asin(x) → el arcoseno de x.
- acos(x) → el arcocoseno de x.
- atan(x) → el arcotangente de x.

Estas funciones toman un argumento (verifican que sea correcto) y devuelven una medida de un ángulo en radianes.

Para trabajar eficazmente con mediciones de ángulos, el módulo math proporciona las siguientes entidades:

- pi → una constante con un valor que es una aproximación de π.
- radians(x) → una función que convierte x de grados a radianes.
- degrees(x) → una función que convierte x de radianes a grados.

El programa de ejemplo no es muy sofisticado, pero ¿puedes predecir sus resultados?

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.)
print(ar == pi / 2.)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

Además de las funciones circulares (enumeradas anteriormente), el módulo math también contiene un conjunto de sus análogos hiperbólicos:

- sinh(x) → el seno hiperbólico.
- cosh(x) → el coseno hiperbólico.
- tanh(x) → la tangente hiperbólico.
- asinh(x) → el arcoseno hiperbólico.
- acosh(x) → el arcocoseno hiperbólico.
- atanh(x) → el arcotangente hiperbólico.

Existe otro grupo de las funciones math relacionadas con la **exponenciación**:

- e → una constante con un valor que es una aproximación del número de Euler (e).
- exp(x) → encontrar el valor de ex.
- log(x) → el logaritmo natural de x.
- log(x, b) → el logaritmo de x con base b.
- log10(x) → el logaritmo decimal de x (más preciso que log(x, 10)).
- log2(x) → el logaritmo binario de x (más preciso que log(x, 2)).

Nota: la función **pow()**:

- pow(x, y) → encuentra el valor de xy (toma en cuenta los dominios).

Esta es una función incorporada y no se tiene que importar.

El último grupo consta de algunas funciones de propósito general como:

- ceil(x) → devuelve el entero más pequeño mayor o igual que x.
- floor(x) → el entero más grande menor o igual que x.
- trunc(x) → el valor de x truncado a un entero (ten cuidado, no es equivalente a ceil o floor).
- factorial(x) → devuelve x! (x tiene que ser un valor entero y no negativo).
- hypot(x, y) → devuelve la longitud de la hipotenusa de un triángulo rectángulo con las longitudes de los catetos iguales a (x) y (y) (lo mismo que sqrt(pow(x, 2) + pow(y, 2)) pero más preciso).

Analiza el programa cuidadosamente.

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
```

```
print(trunc(-x), trunc(-y))
```

Demuestra las diferencias fundamentales entre ceil(), floor() y trunc().

¿Existe aleatoriedad real en las computadoras?

Otro módulo que vale la pena mencionar es el que se llama **random**.

Ofrece algunos mecanismos que permiten operar con **números pseudoaleatorios**.

Toma en cuenta el prefijo **pseudo** - los números generados por los módulos pueden parecer aleatorios en el sentido de que no se pueden predecir, pero no hay que olvidar que todos se calculan utilizando algoritmos muy refinados.

Los algoritmos no son aleatorios, son deterministas y predecibles. Solo aquellos procesos físicos que se salgan completamente de nuestro control (como la intensidad de la radiación cósmica) pueden usarse como fuente de datos aleatorios reales. Los datos producidos por computadoras deterministas no pueden ser aleatorios de ninguna manera.

Un generador de números aleatorios toma un valor llamado **semilla**, lo trata como un valor de entrada, calcula un número «aleatorio» basado en él (el método depende de un algoritmo elegido) y produce una **nueva semilla**.

La duración de un ciclo en el que todos los valores semilla son únicos puede ser muy largo, pero no es infinito: tarde o temprano los valores iniciales comenzarán a repetirse y los valores generadores también se repetirán. Esto es normal. Es una característica, no un error.

El valor de la semilla inicial, establecido durante el inicio del programa, determina el orden en que aparecerán los valores generados.

El factor aleatorio del proceso puede ser **aumentado al establecer la semilla tomando un número de la hora actual** - esto puede garantizar que cada ejecución del programa comience desde un valor semilla diferente (por lo tanto, usará diferentes números aleatorios).

Afortunadamente, Python realiza dicha inicialización al importar el módulo.

Funciones seleccionadas del módulo random

La función random

La función general llamada random() (no debe confundirse con el nombre del módulo) produce un número flotante x entre el rango (0.0, 1.0) - en otras palabras: (0.0 ⇐ x < 1.0).

El programa de ejemplo a continuación producirá cinco valores pseudoaleatorios, ya que sus valores están determinados por el valor semilla actual (bastante impredecible), no puedes adivinarlos:

```
from random import random

for i in range(5):
    print(random())
```

Ejecuta el programa. Esto es lo que tenemos:

```
0.9535768927411208
```

```
0.5312710096244534
0.8737691983477731
0.5896799172452125
0.02116716297022092
salida de muestra
```

La función seed

La función seed() es capaz de directamente establecer la semilla del generador. Te mostramos dos de sus variantes:

- seed() - establece la semilla con la hora actual.
- seed(int_value) - establece la semilla con el valor entero int_value.

Hemos modificado el programa anterior; de hecho, hemos eliminado cualquier rastro de aleatoriedad del código:

```
from random import random, seed

seed(0)

for i in range(5):
    print(random())
```

Debido al hecho de que la semilla siempre se establece con el mismo valor, la secuencia de valores generados siempre se ve igual.

Ejecuta el programa. Esto es lo que tenemos:

```
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

¿Y tú?

Nota: tus valores pueden ser ligeramente diferentes si tu sistema utiliza aritmética de punto flotante más precisa o menos precisa, pero la diferencia se verá bastante lejos del punto decimal.

Las funciones randrange y randint

Si deseas valores aleatorios enteros, una de las siguientes funciones encajaría mejor:

- randrange(fin)
- randrange(inico, fin)
- randrange(inicio, fin, incremento)
- randint(izquierda, derecha)

Las primeras tres invocaciones generarán un valor entero tomado (pseudoaleatoriamente) del rango:

- range(fin)
- range(inicio, fin)
- range(inicio, fin, incremento)

Toma en cuenta la **exclusión implícita del lado derecho**.

La última función es equivalente a randrange(izquierda, derecha+1) - genera el valor entero i, el cual cae en el rango [izquierda, derecha] (sin exclusión en el lado derecho).

Este programa generará una línea que consta de tres ceros y un cero o un uno en el cuarto lugar.

```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1))
```

Las funciones anteriores tienen una desventaja importante: pueden producir valores repetidos incluso si el número de invocaciones posteriores no es mayor que el rango especificado.

Es muy probable que el programa genere un conjunto de números en el que algunos elementos no sean únicos.

```
from random import randint

for i in range(10):
    print(randint(1, 10), end=',')
```

Esto es lo que se obtuvo al ejecutarlo:

```
9,4,5,4,5,8,9,4,8,4,
```

Las funciones choice y sample

Como puedes ver, esta no es una buena herramienta para generar números para la lotería. Afortunadamente, existe una mejor solución que escribir tu propio código para verificar la singularidad de los números «sorteados».

Es una función con el nombre de choice:

- choice(secuencia)
- sample(secuencia, elementos_a_elegir=1)

La primera variante elige un elemento «aleatorio» de la secuencia de entrada y lo devuelve.

El segundo crea una lista (una muestra) que consta del elemento elementos_a_elegir (que por defecto es 1) «sorteado» de la secuencia de entrada.

En otras palabras, la función elige algunos de los elementos de entrada, devolviendo una lista con la elección. Los elementos de la muestra se colocan en orden aleatorio. Nota que elementos_a_elegir no debe ser mayor que la longitud de la secuencia de entrada.

Observa el código a continuación:

```
from random import choice, sample

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(my_list))
print(sample(my_list, 5))
```

```
print(sample(my_list, 10))
```

Nuevamente, la salida del programa no es predecible. Nuestros resultados se ven así:

```
4
[3, 1, 8, 9, 10]
[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]
```

Platform

A veces, puede ser necesario encontrar información no relacionada con Python. Por ejemplo, es posible que necesites conocer la ubicación de tu programa dentro del entorno de la computadora.

Imagina el entorno de tu programa como una pirámide que consta de varias capas o plataformas.



Las capas son:

- El código (en ejecución) se encuentra en la parte superior.
- Python (mejor dicho, su entorno de ejecución) se encuentra directamente debajo de él.
- La siguiente capa de la pirámide se llena con el SO (sistema operativo): el entorno de Python proporciona algunas de sus funcionalidades utilizando los servicios del sistema operativo. Python, aunque es muy potente, no es omnipotente: se ve obligado a usar muchos ayudantes si va a procesar archivos o comunicarse con dispositivos físicos.
- La capa más inferior es el hardware: el procesador (o procesadores), las interfaces de red, los dispositivos de interfaz humana (ratones, teclados, etc.) y toda otra maquinaria necesaria para hacer funcionar la computadora: el sistema operativo sabe como emplearlos y utiliza muchos trucos para trabajar con todas las partes en un ritmo constante.

Esto significa que algunas de las acciones del programa tienen que recorrer un largo camino para ejecutarse con éxito, imagina que:

- **Tu códig**o quiere crear un archivo, por lo que invoca una de las funciones de Python.
- **Python** acepta la orden, la reorganiza para cumplir con los requisitos del sistema operativo local, es como poner el sello «aprobado» en una solicitud y lo envía (esto puede recordarte una cadena de mando).
- El **SO** comprueba si la solicitud es razonable y válida (por ejemplo, si el nombre del archivo se ajusta a algunas reglas de sintaxis) e intenta crear el archivo. Tal operación, aparentemente es muy simple, no es atómica: consiste de muchos pasos menores tomados por:
- El **hardware**, el cual es responsable de activar los dispositivos de almacenamiento (disco duro, dispositivos de estado sólido, etc.) para satisfacer las necesidades del sistema operativo.

Por lo general, no eres consciente de todo ese alboroto: quieres que se cree el archivo y eso es todo.

Pero a veces quieres saber más, por ejemplo, el nombre del sistema operativo que aloja Python y algunas características que describen el hardware que aloja el sistema operativo.

Hay un módulo que proporciona algunos medios para permitir saber dónde se encuentra y qué componentes funcionan. El módulo se llama platform. Veamos algunas de las funciones que brinda para ti.

Funciones seleccionadas del módulo platform

La función platform

El módulo **platform** permite acceder a los datos de la plataforma subyacente, es decir, hardware, sistema operativo e información sobre la versión del intérprete.

Existe también una función que puede mostrar todas las capas subyacentes en un solo vistazo, llamada platform. Simplemente devuelve una cadena que describe el entorno; por lo tanto, su salida está más dirigida a los humanos que al procesamiento automatizado (lo verás pronto).

Así es como se puede invocar:

```
platform(aliased = False, terse = False)
```

Y ahora:

- aliased → cuando se establece a True (o cualquier valor distinto a cero) puede hacer que la función presente los nombres de capa subyacentes alternativos en lugar de los comunes.
- terse → cuando se establece a True (o cualquier valor distinto a cero) puede convencer a la función de presentar una forma más breve del resultado (si lo fuera posible).

Ejecutamos el programa usando tres plataformas diferentes, esto es lo que se obtuvo:

Intel x86 + Windows ® Vista (32 bit):

```
Windows-Vista-6.0.6002-SP2
Windows-Vista-6.0.6002-SP2
Windows-Vista
```

Intel x86 + Gentoo Linux (64 bit):

```
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_@_2.20GHz-with-glibc2.3.4
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9
```

También puedes ejecutar el programa en el IDLE de tu máquina local para verificar que salida tendrá:

```
from platform import platform

print(platform())
print(platform(1))
print(platform(0, 1))
```

La función machine

En ocasiones, es posible que solo se desee conocer el nombre genérico del procesador que ejecuta el sistema operativo junto con Python y el código, una función llamada **machine()** te lo dirá. Como anteriormente, la función devuelve una cadena.

Nuevamente, ejecutamos el programa en tres plataformas diferentes:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
x86_64
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

```
from platform import machine

print(machine())
```

La función processor

La función **processor()** devuelve una cadena con el nombre real del procesador (si lo fuese posible).

Una vez más, ejecutamos el programa en tres plataformas diferentes:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

```
from platform import processor

print(processor())
```

La función system

Una función llamada **system()** devuelve el nombre genérico del sistema operativo en una cadena.

Nuestras plataformas de ejemplo se presentan de la siguiente manera:

Intel x86 + Windows ® Vista (32 bit):

```
Windows
```

Intel x86 + Gentoo Linux (64 bit):

```
<code>
```

```
Linux
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux
```

```
from platform import system

print(system())
```

La función version

La versión del sistema operativo se proporciona como una cadena por la función version().

Ejecuta el código y verifica su salida. Esto es lo que tenemos:

Intel x86 + Windows ® Vista (32 bit):

```
6.0.6002
```

Intel x86 + Gentoo Linux (64 bit):

```
#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)
```

```
from platform import version

print(version())
from platform import version
```

Las funciones python_implementation y python_version_tuple

Si necesitas saber que versión de Python está ejecutando tu código, puedes verificarlo utilizando una serie de funciones dedicadas, aquí hay dos de ellas:

- python_implementation() → devuelve una cadena que denota la implementación de Python (espera CPython aquí, a menos que decidas utilizar cualquier rama de Python no canónica).
- python_version_tuple() → devuelve una tupla de tres elementos la cual contiene:
    - La parte mayor de la versión de Python.
    - La parte menor.
    - El número del nivel de parche.

```
from platform import python_implementation, python_version_tuple

print(python_implementation())

for atr in python_version_tuple():
    print(atr)
```

Nuestro programa de ejemplo produjo el siguiente resultado:

```
CPython
3
7
7
```

Es muy probable que tu versión de Python sea diferente.

Índice de Módulos de Python

Aquí solo hemos cubierto los conceptos básicos de los módulos de Python. Los módulos de Python conforman su propio universo, en el que Python es solo una galaxia, y nos aventuraríamos a decir que explorar las profundidades de estos módulos puede llevar mucho más tiempo que familiarizarse con Python «puro».

Además, la comunidad de Python en todo el mundo crea y mantiene cientos de módulos adicionales utilizados en aplicaciones muy específicas como la genética, la psicología o incluso la astrología.

Estos módulos no están (y no serán) distribuidos junto con Python, o a través de canales oficiales, lo que hace que el universo de Python sea más amplio, casi infinito.

Puedes leer sobre todos los módulos estándar de Python aquí: https://docs.python.org/3/py-modindex.html.

No te preocupes, no necesitarás todos estos módulos. Muchos de ellos son muy específicos.

Todo lo que se necesita hacer es encontrar los módulos que se desean y aprender a cómo usarlos. Es fácil.

Puntos Clave

1. Una función llamada dir() puede mostrarte una lista de las entidades contenidas dentro de un módulo importado. Por ejemplo:

```
import os
dir(os)
```

Imprime una lista de todo el contenido del módulo os el cual, puedes usar en tu código.

2. El módulo math contiene más de 50 funciones y constantes que realizan operaciones matemáticas (como sine(), pow(), factorial()) o aportando valores importantes (como π y la constante de Euler e).

3. El módulo random agrupa más de 60 entidades diseñadas para ayudarte a usar números pseudoaleatorios. No olvides el prefijo «pseudo», ya que no existe un número aleatorio real cuando se trata de generarlos utilizando los algoritmos de la computadora.

4. El módulo platform contiene alrededor de 70 funciones que te permiten sumergirte en las capas subyacentes del sistema operativo y el hardware. Usarlos te permite aprender más sobre el entorno en el que se ejecuta tu código.

5. El Índice de Módulos de Python (https://docs.python.org/3/py-modindex.html es un directorio de módulos impulsado por la comunidad disponible en el universo de Python. Si deseas encontrar un módulo que se adapte a tus necesidades, comienza tu búsqueda allí.

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

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

Last update: **23/06/2022 11:21**