PROGRAMACIÓN II - UNIDAD 3

Ing. Gastón Weingand (gaston.weingand@uai.edu.ar)

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

Agenda

1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

5-Generadores y decoradores

Generadores

Las expresiones generadoras funcionan de forma muy similar a la comprensión de listas. De hecho, su sintaxis es exactamente igual, a excepción de que se utilizan paréntesis en lugar de corchetes:

```
l2 = (n ** 2 for n in l)
```

5-Generadores y decoradores

Generadores

Sin embargo, las expresiones generadoras se diferencian de la comprensión de listas en que no se devuelve una lista, sino un generador.

```
l = [0, 1, 2, 3]
l2 = [n ** 2 for n in l] #Comprensión de listas
print(l2)

l2 = (n ** 2 for n in l) #Generador
print(l2)
```

```
[0, 1, 4, 9]
<generator object <genexpr> at 0x0000027959DD85C8>
```

5-Generadores y decoradores

Generadores

Un generador es una clase especial de función que genera valores sobre los que iterar. Para devolver el siguiente valor sobre el que iterar se utiliza la palabra clave yield en lugar de return. Veamos por ejemplo un generador que devuelva números de n a m con un salto s.

```
def mi_generador(n, m, s):
    while(n <= m):
        yield n
        n += s

for n in mi_generador(0, 5, 1):
    print (n)

lista = list(mi_generador(0,5,1))
print(lista)
```

El generador se puede utilizar en cualquier lugar donde se necesite un objeto iterable. También puede asignarse a list directamente

5-Generadores y decoradores

Decoradores

Un decorador no es mas que una función que recibe una función como parámetro y devuelve otra función como resultado. Por ejemplo, podríamos querer añadir la funcionalidad de que se imprimiera el nombre de la función llamada por motivos de depuración:

```
#Decoradores
def mi_decorador(funcion):
    def nueva(*args):
        print ("Llamada a la funcion"), funcion.__name__
        retorno = funcion(*args)
        return retorno
    return nueva
```

5-Generadores y decoradores

Decoradores

Decoro la función imp con mi_decorador

```python
#Decoradores
def mi_decorador(funcion):
    def nueva(*args):
        print ("Llamada a la funcion"), funcion.__name__
        retorno = funcion(*args)
        return retorno
    return nueva

@mi_decorador #mi_decorador(imp)("hola") #Da el mismo resultado...
def imp(s):
    print(s)

imp("Hola")
```

5-Generadores y decoradores

Decoradores

```
#Si quisiéramos aplicar más de un decorador bastaría añadir una nueva
#línea con el nuevo decorador. Es importante advertir que los decoradores
# se ejecutarán de abajo hacia arriba
@otro_decorador
@mi_decorador
def imp(s):
    print (s)
```

Agenda

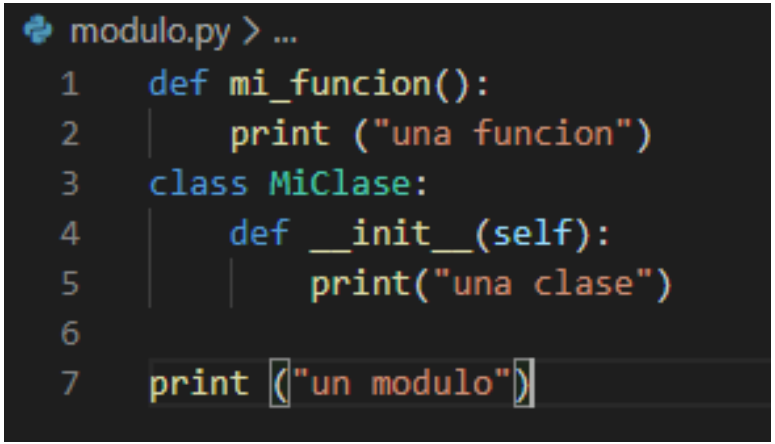
1. Funciones de orden superior.
2. Iteraciones sobre listas.
3. Funciones lambda.
4. Comprensión de listas.
5. Generadores y decoradores.
6. Módulos y paquetes. Excepciones.
7. Ejemplos de su utilización. Integración con clases y objetos.

6-Módulos

Para facilitar el mantenimiento y la lectura los programas demasiado largos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica de nuestro código. Los ficheros son su contrapartida física: cada archivo Python almacenado en disco equivale a un módulo.

6-Módulos

Vamos a crear nuestro primer módulo entonces creando un pequeño archivo modulo.py con el siguiente contenido:



6-Módulos

Si quisiéramos utilizar la funcionalidad definida en este módulo en nuestro programa tendríamos que importarlo. Para importar un módulo se utiliza la palabra clave import seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión.

```
import modulo
modulo.mi_funcion()
clase = modulo.MiClase()
```

6-Módulos

Sin embargo, es posible utilizar la construcción from-import para ahorrarnos el tener que indicar el nombre del módulo antes del objeto que nos interesa. De esta forma se importa el objeto o los objetos que indiquemos al espacio de nombres actual.

```
from modulo import *
a = MiClase()
mi_funcion()
```

En el caso de que Python no encontrara ningún módulo con el nombre especificado, se lanzaría una excepción de tipo ImportError.

6-Módulos

Por último, es interesante comentar que en Python los módulos también son objetos; de tipo module en concreto. Por supuesto esto significa que pueden tener atributos y métodos. Uno de sus atributos, __name__, se utiliza a menudo para incluir código ejecutable en un módulo pero que este sólo se ejecute si se llama al módulo como programa, y no al importarlo. Para lograr esto basta saber que cuando se ejecuta el módulo directamente __name__ tiene como valor "__main__", mientras que cuando se importa, el valor de __name__ es el nombre del módulo:

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

6-Módulos

Otro atributo interesante es __doc__, que, como en el caso de funciones y clases, sirve a modo de documentación del objeto (docstring o cadena de documentación). Su valor es el de la primera línea del cuerpo del módulo, en el caso de que esta sea una cadena de texto; en caso contrario valdrá None

modulo.py > ...

```
1   """Definición del módulo, puedo comentar qué es lo que hace..."""
2
```

```
import modulo
modulo.mi_funcion()
clase
```

module modulo

Definición del módulo, puedo comentar qué es lo que hace...

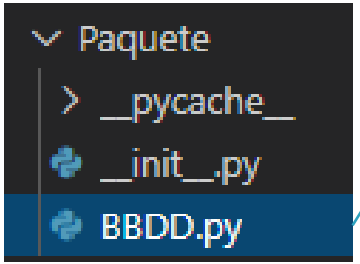
6-Paquetes

Si los módulos sirven para organizar el código, los paquetes sirven para organizar los módulos. Los paquetes son tipos especiales de módulos (ambos son de tipo module) que permiten agrupar módulos relacionados. Mientras los módulos se corresponden a nivel físico con los archivos, los paquetes se representan mediante directorios

En una aplicación cualquiera podríamos tener, por ejemplo, un paquete IU para la interfaz o un paquete BBDD para la persistencia a base de datos.

6-Paquetes

Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo __init__.py en dicha carpeta. En este archivo se pueden definir elementos que pertenezcan a dicho paquete, como una constante DRIVER para el paquete BBDD, aunque habitualmente se tratará de un archivo vacío. Para hacer que un cierto módulo se encuentre dentro de un paquete, basta con copiar el archivo que define el módulo al directorio del paquete.



Genero estructura de carpeta

```
Paquete > BBDD.py
1   def test():
2       print("Hello")
```

```
from Paquete.BBDD import test
test()
```

Importo el paquete.modulo donde lo necesite

6-Excepciones

Las excepciones son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como el intentar dividir un número entre 0 o el intentar acceder a un archivo que no existe, este genera o lanza una excepción, informando al usuario de que existe algún problema.

6-Excepciones

Veamos un pequeño programa que lanzaría una excepción al intentar dividir 1 entre 0.

```
#Excepciones...
def division(a, b):
    return a / b
def calcular():
    division(1, 0)

calcular()
```

6-Excepciones

Si lo ejecutamos obtendremos el siguiente mensaje de error:

```
Exception has occurred: ZeroDivisionError
division by zero
```

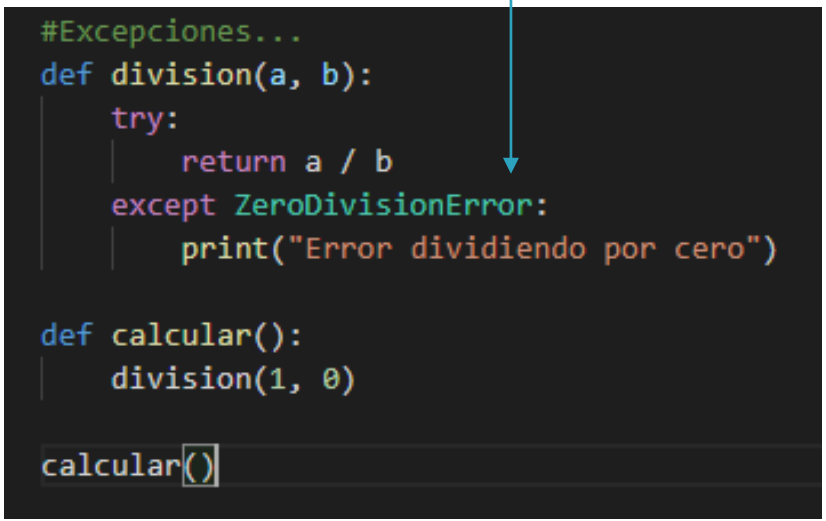
Además del error veremos el trazado de pila o traceback, que consiste en una lista con las llamadas que provocaron la excepción. Línea por línea, función a función

```
File "D:\Facultad\Programacion II\Python\Material Cursada\Codigo\dia3-3.py", line 3, in division
  return a / b
File "D:\Facultad\Programacion II\Python\Material Cursada\Codigo\dia3-3.py", line 6, in calcular
  division(1, 0)
File "D:\Facultad\Programacion II\Python\Material Cursada\Codigo\dia3-3.py", line 8, in <module>
  calcular()
```

6-Excepciones

En Python se utiliza una construcción try-except para capturar y tratar las excepciones. El bloque try (Intentar) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque except (Excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. Muchas veces nuestro tratamiento de la excepción consistirá simplemente en imprimir un mensaje más amigable para el usuario, otras veces nos interesará registrar los errores y de vez en cuando podremos establecer una estrategia de resolución del problema.

Puede dejarse solo except para atajar cualquier error



```python
#Excepciones...
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error dividiendo por cero")

def calcular():
    division(1, 0)

calcular()
```

6-Excepciones

Además, podemos hacer que un mismo except sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

```
try:
    num = int("3a")
    print ("paso por acá")
except (NameError, ValueError):
    print ("Ocurrio un error")
```

NameError: Una variable local o global no se encuentra
ValueError: Un parámetro no es del tipo esperado...

6-Excepciones

La construcción try-except puede contar además con una clausula else, que define un fragmento de código a ejecutar sólo si no se ha producido ninguna excepción en el try.

```
try:
    num = 33
except (NameError, ValueError):
    print ("Ocurrio un error")
else:
    print("Todo está bien")
```

6-Excepciones

También existe una clausula finally que se ejecuta siempre, se produzca o no una excepción. Esta clausula se suele utilizar, entre otras cosas, para tareas de limpieza.

```
try:
    z = 10 / 0
except ZeroDivisionError:
    print ("Division por cero")
finally:
    print("Limpiando")
```

6-Excepciones

También es interesante comentar que como programadores podemos crear y lanzar nuestras propias excepciones. Basta crear una clase que herede de Exception o cualquiera de sus hijas y lanzarla con raise.

```
class MiError(Exception):
    def __init__(self, valor):
        self.valor = valor

    def __str__(self):
        return "Error " + str(self.valor)

try:
    if 40 > 20:
        raise MiError(33)
except MiError as e:
    print (e)
```

6-Excepciones

Algunas excepciones clásicas:

- **BaseException:** Clase de la que heredan todas las excepciones.
- **Exception(BaseException):** Super clase de todas las excepciones que no sean de salida. GeneratorExit(Exception): Se pide que se salga de un generador. StandardError(Exception): Clase base para todas las excepciones que no tengan que ver con salir del intérprete.
- **ArithmeticError(StandardError):** Clase base para los errores aritméticos. FloatingPointError(ArithmeticError): Error en una operación de coma flotante. OverflowError(ArithmeticError): Resultado demasiado grande para poder representarse.
- **ZeroDivisionError(ArithmeticError):** Lanzada cuando el segundo argumento de una operación de división o módulo era 0.

7-Ejemplos

Emulando métodos abstractos…

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("check_ripeness method not implemented!")

class Apple(Fruit):
    pass

try:
    a = Apple()
    a.check_ripeness() # Lanzo una excepción porque Apple no implementó el método check...
except Exception as e:
    print(e)
```

7-Ejemplos

Clases abstractas…

```python
from abc import ABCMeta, abstractmethod

class AbstractClass(metaclass=ABCMeta): #Indico a la clase que es abstracta

    #Decoro los métodos abstractos
    @abstractmethod
    def metodo_virtual_que_la_clase_hija_debe_implementar(self):
        pass

    def otro_metodo(self):
        print("Implementación...")

class SubClass(AbstractClass):
    def metodo_virtual_que_la_clase_hija_debe_implementar(self):
        return "Hello"

a = SubClass()
print(a.metodo_virtual_que_la_clase_hija_debe_implementar())
a.otro_metodo()
```