Taller de Python: continuación

Posma Group

8 de febrero de 2014

Contenido

1. Introducción
2. Modelo de ejecución
3. Estructura de un proyecto
4. Programación Orientada a Objetos
5. Módulos nativos
6. Decoradores
7. Iterables e iteradores

Continuamos estudiando...

En la primera clase hemos cubierto lo básico para entender de manera formal las particularidades del lenguaje Python, sin embargo, aún no ahondamos en las áreas de mayor aplicación práctica del lenguaje. En esta oportunidad exploraremos las características más avanzadas de Python.

Todo es un objeto

Aunque no es un concepto fácil de entender, para el intérprete de Python todas las cosas (tipos, valores, funciones, módulos, etc) son objetos.

Todo es un objeto

Aunque no es un concepto fácil de entender, para el intérprete de Python todas las cosas (tipos, valores, funciones, módulos, etc) son objetos.

Hagamos esta prueba:

```
>>> n = 7
>>> dir(n)
```

Todo es un objeto

De igual manera, las funciones que definimos en nuestro programa, podemos explorarlas mediante el uso de dir. Incluso podemos explorar la misma función `dir`.

Todo es un objeto

De igual manera, las funciones que definimos en nuestro programa, podemos explorarlas mediante el uso de dir. Incluso podemos explorar la misma función `dir`.

```
>>> dir(dir)
```

Todo es un objeto

De igual manera, las funciones que definimos en nuestro programa, podemos explorarlas mediante el uso de dir. Incluso podemos explorar la misma función `dir`.

```
>>> dir(dir)
```

...Podemos notar que realmente, hasta las funciones nativas son objetos.

Todo es un objeto

De igual manera, las funciones que definimos en nuestro programa, podemos explorarlas mediante el uso de dir. Incluso podemos explorar la misma función `dir`.

```
>>> dir(dir)
```

...Podemos notar que realmente, hasta las funciones nativas son objetos.

Otra cosa que podemos probar, es invocar la función `dir()` sin argumentos.

Nombres y asociaciones

En Python, es importante entender que las "variables"son sólo nombres, los cuales están asociados siempre a algún objeto.

Nombres y asociaciones

En Python, es importante entender que las "variables"son sólo nombres, los cuales están asociados siempre a algún objeto.

Para saber el identificador del objeto al cual una variable está asociada, utilizamos la función `id()`:

```
>>> n = 7
>>> id(n)
36031448
>>> id(7)
36031448
```

Nombres y asociaciones

Incluso los valores constantes como `None`, `True` y `False`, en lugar de ser palabras reservadas del lenguaje, son objetos globales.

```
>>> id(None)
8696288
>>> id(True)
8696208
>>> id(False)
8696240
```

Nombres y asociaciones

Para saber si dos nombres se refieren a un mismo objeto, se utiliza el operador `is`.

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L3 = [1, 2, 3]
>>> L1 is L2
True
>>> L1 is L3
False
>>> L1 == L3
True
```

Nombres y asociaciones

Es incluso factible asociar el nombre de una función a una nueva función o a algún otro valor:

```
>>> def funcion_nula():
...     pass
...
>>> funcion_nula = 42
>>> incr = funcion_nula
>>> incr
42
```

Nombres y asociaciones

Es incluso factible asociar el nombre de una función a una nueva función o a algún otro valor:

```
>>> def funcion_nula():
...     pass
...
>>> funcion_nula = 42
>>> incr = funcion_nula
>>> incr
42
```

Para el intérprete sólo se trata de nombres y objetos.

Funciones: objetos de primer orden

Como ya hemos mencionado, incluso las funciones son consideradas objetos para el intérprete de Python.

Funciones: objetos de primer orden

Como ya hemos mencionado, incluso las funciones son consideradas objetos para el intérprete de Python.

Esto significa que es posible pasar funciones como argumentos a otras funciones:

```
>>> def map(funcion, secuencia):
...     r = []
...     for e in secuencia:
...         r.append(f(e))
...     return r
...
>>> def incr(n):
...     return n + 1
...
>>> map(incr, [1,2,3,4])
[2, 3, 4, 5]
```

Mutabilidad

Como hemos mencionado anteriormente, existen dos tipos de objetos: mutables e inmutables. Las cadenas de texto, por ejemplo, son inmutables.

Mutabilidad

Como hemos mencionado anteriormente, existen dos tipos de objetos: mutables e inmutables. Las cadenas de texto, por ejemplo, son inmutables.

```
>>> c = "hola mundo"
>>> c[0] = "B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Mutabilidad

Como hemos mencionado anteriormente, existen dos tipos de objetos: mutables e inmutables. Las cadenas de texto, por ejemplo, son inmutables.

```
>>> c = "hola mundo"
>>> c[0] = "B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

De igual manera son inmutables las tuplas y los números.

Mutabilidad

...Sin embargo:

```
>>> L = [1, 2, 3]
>>> tup = (L, L)
>>> tup
([1, 2, 3], [1, 2, 3])
>>> L.remove(3)
>>> tup
([1, 2], [1, 2])
```

Estructura de un proyecto

- Al momento de elaborar un proyecto de cierta complejidad, es indispensable organizar los archivos y sus dependencias de manera lógica, eficiente e intuitiva.

Estructura de un proyecto

- Al momento de elaborar un proyecto de cierta complejidad, es indispensable organizar los archivos y sus dependencias de manera lógica, eficiente e intuitiva.
- A pesar de que Python provee un sistema de módulos y dependencias bastante sencillo de utilizar, queda de parte del programador definir el proyecto con buenas prácticas.

Módulos

- Los módulos son la abstracción organizativa más básica del lenguaje (un archivo .py).

Módulos

- Los módulos son la abstracción organizativa más básica del lenguaje (un archivo .py).
- El objetivo de un módulo es abarcar diversas funciones, clases y otros objetos conceptualmente relacionados.

Módulos

- Los módulos son la abstracción organizativa más básica del lenguaje (un archivo .py).
- El objetivo de un módulo es abarcar diversas funciones, clases y otros objetos conceptualmente relacionados.
- Debe seguirse siempre el principio básico de “alto acoplamiento, baja cohesión”.

Módulos

- Los módulos son la abstracción organizativa más básica del lenguaje (un archivo .py).
- El objetivo de un módulo es abarcar diversas funciones, clases y otros objetos conceptualmente relacionados.
- Debe seguirse siempre el principio básico de “alto acoplamiento, baja cohesión”.
- Es recomendable que los nombres de los módulos sean simples, completamente en minúscula y sin caracteres especiales.

Módulos

Cuando hacemos `import math`, por ejemplo, estamos importando el módulo `math` en nuestro archivo fuente.

Módulos

Cuando hacemos `import math`, por ejemplo, estamos importando el módulo `math` en nuestro archivo fuente.

Esto no significa que el código del módulo en cuestión es completamente copiado en nuestro código. Más bien el código del módulo se mantiene aislado en su propio espacio de nombres.

Módulos

Cuando hacemos `import math`, por ejemplo, estamos importando el módulo `math` en nuestro archivo fuente.

Esto no significa que el código del módulo en cuestión es completamente copiado en nuestro código. Más bien el código del módulo se mantiene aislado en su propio espacio de nombres.

```
import math

def sqrt():
    print "hello world"

def raiz_cuadrada(n):
    return math.sqrt(n)
```

Módulos

Si queremos importar una función en particular sin necesitar del espacio de nombres, podemos hacer:

```
from math import sqrt
```

Módulos

Si queremos importar una función en particular sin necesitar del espacio de nombres, podemos hacer:

```
from math import sqrt
```

También es posible importar miembros de un módulo bajo un nombre nuevo, de la siguiente manera:

```
from math import sqrt as raiz
```

Paquetes

- Python provee un sistema sencillo de empaquetado, que es simplemente una extensión del mecanismo de módulos pero para directorios.
- La finalidad de un paquete es colocar en una carpeta uno o más módulos funcionalmente relacionados.

Paquetes

- Python provee un sistema sencillo de empaquetado, que es simplemente una extensión del mecanismo de módulos pero para directorios.
- La finalidad de un paquete es colocar en una carpeta uno o más módulos funcionalmente relacionados.
- Para que una carpeta sea considerada un paquete, debe contener un archivo `__init__.py`. En el cual pueden definirse los objetos que serán comunes al paquete.

```
import paquete

def mi_funcion():
    return paquete.modulo.otra_funcion()
```

Buenas prácticas

Renombramiento dinámico

Es recomendable evitar en la medida de lo posible el anidamiento excesivo de paquetes dentro de otros paquetes.

Buenas prácticas

Renombramiento dinámico

Es recomendable evitar en la medida de lo posible el anidamiento excesivo de paquetes dentro de otros paquetes.

Sin embargo, algunos niveles de complejidad hacen inevitable el anidamiento de paquetes. En este caso:

```
import paquete.otropaquete.otropaquetemas.modulo as mod

mod.funcion()
```

Buenas prácticas

- Evitar las dependencias circulares

Buenas prácticas

- Evitar las dependencias circulares
- Evitar el "código espagueti"

Buenas prácticas

- Evitar las dependencias circulares
- Evitar el "código espagueti"
- Seguir las convenciones del PEP-8 (http://www.python.org/dev/peps/pep-0008/)

Programación Orientada a Objetos

Python es un lenguaje que provee todas las características del paradigma Orientado a Objetos, pero sin obligar al programador a adoptar dicho paradigma.

Programación Orientada a Objetos

Python es un lenguaje que provee todas las características del paradigma Orientado a Objetos, pero sin obligar al programador a adoptar dicho paradigma.

De hecho, la manera en la que Python maneja los módulos y los espacios de nombres, hace posible el encapsulamiento y la separación de capas de abstracción sin necesidad del uso de clases.

Programación Orientada a Objetos

Python es un lenguaje que provee todas las características del paradigma Orientado a Objetos, pero sin obligar al programador a adoptar dicho paradigma.

De hecho, la manera en la que Python maneja los módulos y los espacios de nombres, hace posible el encapsulamiento y la separación de capas de abstracción sin necesidad del uso de clases.

Es por esto que muchos programadores de Python suelen no recurrir a la Programación Orientada a Objetos a menos que sea absolutamente necesario.

Clases

Las clases en Python se definen con la palabra reservada class, seguida de un nombre y opcionalmente las clases de las cuales ésta pueda heredar.

```
class MiClase:
    pass
```

Clases

Las clases en Python se definen con la palabra reservada class, seguida de un nombre y opcionalmente las clases de las cuales ésta pueda heredar.

```
class MiClase:
    pass
```

En este caso hemos definido una clase que no define ningún miembro. Lo único que requiere una clase es un nombre.

Herencia

Como es común en los lenguajes Orientados a Objetos, una clase puede "heredar" los atributos y métodos de otra clase.

```
class ClaseHija(paquete.modulo.ClasePadre):
    pass
```

Herencia

Como es común en los lenguajes Orientados a Objetos, una clase puede "heredar" los atributos y métodos de otra clase.

```
class ClaseHija(paquete.modulo.ClasePadre):
    pass
```

La herencia de clases, como en la mayoría de los lenguajes OO, hace que las instancias de las subclases tengan acceso a los atributos de las superclases.

Herencia

Como es común en los lenguajes Orientados a Objetos, una clase puede "heredar" los atributos y métodos de otra clase.

```
class ClaseHija(paquete.modulo.ClasePadre):
    pass
```

La herencia de clases, como en la mayoría de los lenguajes OO, hace que las instancias de las subclases tengan acceso a los atributos de las superclases.

En caso de que la clase hija sobrescriba un método, no implica que esta referencia se pierda; desde la subclase es posible invocar explícitamente métodos de la clase padre mediante la función `super`:

```
super(ClaseHija, self).metodoPadre(argumentos)
```

Herencia

isinstance()

```
>>> isinstance(c, Contador)
True
>>> isinstance(7, int)
True
```

Herencia

isinstance()

```
>>> isinstance(c, Contador)
True
>>> isinstance(7, int)
True
```

issubclass()

```
>>> class ContadorEspecial(Contador):
...     pass
...
>>> issubclass(ContadorEspecial, Contador)
True
```

Herencia múltiple

En Python es posible heredar de más de una clase. Para esto, sencillamente se escriben los nombres de las clases base como argumentos de la clase actual, separados por coma.

Herencia múltiple

En Python es posible heredar de más de una clase. Para esto, sencillamente se escriben los nombres de las clases base como argumentos de la clase actual, separados por coma.

```
class MiClase(ClaseBase1, ClaseBase2, ClaseBase2):
        pass
```

Herencia múltiple

En Python es posible heredar de más de una clase. Para esto, sencillamente se escriben los nombres de las clases base como argumentos de la clase actual, separados por coma.

```
class MiClase(ClaseBase1, ClaseBase2, ClaseBase2):
        pass
```

Al momento de resolver la ubicación de un nombre, el intérprete primero busca en el ámbito local (MiClase), luego busca la definición en ClaseBase1, luego ClaseBase2 y así sucesivamente.

Constructor

- En las clases de Python, llamamos "atributos" a todos los nombres definidos en el espacio de nombres de una clase, bien sea que éstos representen variables o métodos.
- Un atributo particular en las clases es el método `__init__`, que funciona como una especie de "constructor" al momento de instanciar dicha clase.

Constructor

- En las clases de Python, llamamos "atributos" a todos los nombres definidos en el espacio de nombres de una clase, bien sea que éstos representen variables o métodos.
- Un atributo particular en las clases es el método `__init__`, que funciona como una especie de "constructor" al momento de instanciar dicha clase.

```
class Usuario(Persona):
    def __init__(self, nombre, edad):
                self.nombre = nombre
                self.edad = edad
```

Instanciación

Para crear un objeto instancia de la clase `Usuario`, invocamos a la clase directamente, como si fuese una función.

```
>>> u = Usuario("Carlos", 28)
```

Instanciación

Para crear un objeto instancia de la clase `Usuario`, invocamos a la clase directamente, como si fuese una función.

```
>>> u = Usuario("Carlos", 28)
```

Esto crea una instancia de Usuario, y automáticamente ejecuta el cuerpo de `__init__`, con los argumentos dados.

Instanciación

Para crear un objeto instancia de la clase `Usuario`, invocamos a la clase directamente, como si fuese una función.

```
>>> u = Usuario("Carlos", 28)
```

Esto crea una instancia de Usuario, y automáticamente ejecuta el cuerpo de `__init__`, con los argumentos dados.

El método `__init__`, y cualquier otro método a ser usado por las instancias de una clase, debe recibir al menos el parámetro `self`.

Instanciación

Para crear un objeto instancia de la clase `Usuario`, invocamos a la clase directamente, como si fuese una función.

```
>>> u = Usuario("Carlos", 28)
```

Esto crea una instancia de Usuario, y automáticamente ejecuta el cuerpo de `__init__`, con los argumentos dados.

El método `__init__`, y cualquier otro método a ser usado por las instancias de una clase, debe recibir al menos el parámetro `self`.

Las variables de instancia no necesitan ser declaradas.

Modificación dinámica de objetos

```
>>> u = Usuario("John", 42)
>>> u.edad
42
u.direccion
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Usuario instance has no attribute 'direccion'
>>> u.direccion = "Chacao"
>>> u.direccion
'Chacao'
```

Destructor

Lo mismo aplica para la destrucción de instancias en Python, no existe realmente un "destructor", pero contamos con un método similar a un destructor: `__del__`.

```
class Saludo:
    def __init__(self, nombre):
        self.nombre = nombre

        def __del__(self):
        print "Adios!..."

        def decir_hola(self):
        print "Hola %s!" % self.nombre
```

Atributos de Clase

Las variables declaradas explícitamente en una clase sin estar ligadas a `self` son consideradas atributos propios de la clase.

```
class MiClase():
    MAX_VALUE = 5000     # se puede utilizar para declarar constantes
```

Atributos de Clase

Las variables declaradas explícitamente en una clase sin estar ligadas a `self` son consideradas atributos propios de la clase.

```
class MiClase():
    MAX_VALUE = 5000     # se puede utilizar para declarar constantes


>>> MiClase.MAX_VALUE
5000
>>> obj = MiClase()
>>> obj.MAX_VALUE
5000
>>> obj.MAX_VALUE = "cadena"
>>> obj.MAX_VALUE
'cadena'
>>> MiClase.MAX_VALUE
5000
```

Atributos de Clase

Es posible para una instancia modificar el valor de un atributo de la clase:

Atributos de Clase

Es posible para una instancia modificar el valor de un atributo de la clase:

```
>>> class Contador():
...     cont = 0
...     def __init__(self):
...         self.__class__.cont += 1
...
>>> Contador.cont
0
>>> c = Contador()
>>> Contador.cont
1
>>> d = Contador()
>>> Contador.cont
2
>>> c.cont
2
```

Atributos de Clase

Es posible para una instancia modificar el valor de un atributo de la clase:

```
>>> class Contador():
...     cont = 0
...     def __init__(self):
...         self.__class__.cont += 1
...
>>> Contador.cont
0
>>> c = Contador()
>>> Contador.cont
1
>>> d = Contador()
>>> Contador.cont
2
>>> c.cont
2
```

Encapsulamiento

- En Python no existe un mecanismo como tal para el encapsulamiento de atributos.
- El encapsulamiento se logra mediante convenciones de nombramiento.

Encapsulamiento

- En Python no existe un mecanismo como tal para el encapsulamiento de atributos.
- El encapsulamiento se logra mediante convenciones de nombramiento.

```
class Encapsulamiento(object):
    def __init__(self, a, b, c):
        self.publico = a
        self._protegido = b
        self.__privado = c
```

Una de las filosofías de Python, es que es un lenguaje “con baterías incluidas”, esto quiere decir que sin necesidad de instalar paquetes externos, ya el lenguaje provee una gama de funcionalidades de alta utilidad en diversas áreas.

Una de las filosofías de Python, es que es un lenguaje “con baterías incluidas”, esto quiere decir que sin necesidad de instalar paquetes externos, ya el lenguaje provee una gama de funcionalidades de alta utilidad en diversas áreas.

La lista de módulos provistos por la biblioteca estándar de Python es muy extensa:

Referencia

http://docs.python.org/2/library/

Módulos nativos

datetime / time

```
>>> import time
>>> from datetime import date
>>> hoy = date.today()
>>> hoy
datetime.date(2007, 12, 5)
>>> mi_cumple = date(hoy.year, 6, 24)
>>> if mi_cumple < hoy:
...     mi_cumple = mi_cumple.replace(year=hoy.year + 1)
>>> mi_cumple
datetime.date(2008, 6, 24)
>>> tiempo_hasta_cumple = abs(mi_cumple - hoy)
>>> print "Quedan %s dias para mi cumple!!!" % tiempo_hasta_cumple.days
'Quedan 202 dias para mi cumple!!!'
```

Módulos nativos

os

El módulo os provee funcionalidades referentes al sistema operativo, como exploración y manipulación de rutas en el sistema de archivos, y manejo de procesos.

Módulos nativos

os

El módulo os provee funcionalidades referentes al sistema operativo, como exploración y manipulación de rutas en el sistema de archivos, y manejo de procesos.

```
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

Módulos nativos

json

Este módulo permite la codificación de estructuras de Python en formato JSON (Javascript Object Notation).

```
>>> import json
>>> L = ['foo', {'bar': ('baz', None, 1.0, 2)}]
>>> json.dumps(L)
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
```

Módulos nativos

random

```
>>> import random
>>> random.randrange(100)
72
>>> random.randrange(100)
7
>>> random.randrange(100)
40
```

Módulos nativos

random

```
>>> import random
>>> random.randrange(100)
72
>>> random.randrange(100)
7
>>> random.randrange(100)
40
```

```
>>> L = range(10)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle(L)
[1, 4, 3, 0, 7, 5, 6, 9, 2, 8]
```

Decoradores

Definición

Un decorador no es más que una función o una clase que envuelve (o decora) otra función o método. La función decorada reemplaza la función original.

Decoradores

Definición

Un decorador no es más que una función o una clase que envuelve (o decora) otra función o método. La función decorada reemplaza la función original.

```
def mi_decorador(func)
    # manipular func
    return func

@mi_decorador
def funcion():
    # Hacer algo
    pass
# funcion() ha sido decorada
```

Decoradores

Los decoradores son útiles para separar comportamientos que son ajenos a la lógica de la función como tal.

Decoradores

Los decoradores son útiles para separar comportamientos que son ajenos a la lógica de la función como tal.

Una función puede ser objeto de más de un decorador:

```
@decorador3
@decorador2
@decorador1
def function():
    # Hacer algo
    pass
```

Decoradores

Los decoradores son útiles para separar comportamientos que son ajenos a la lógica de la función como tal.

Una función puede ser objeto de más de un decorador:

```
@decorador3
@decorador2
@decorador1
def function():
    # Hacer algo
    pass
```

En este caso, el orden de aplicación es desde abajo hacia arriba, comenzando por `@decorador1`.

Decoradores

Un decorador puede implementarse como una función, o como una clase siempre y cuando ésta implemente el método `__call__`.

Decoradores

Un decorador puede implementarse como una función, o como una clase siempre y cuando ésta implemente el método `__call__`.

```
class cached(object):
    def __init__(self, func):
        self.func = func
        self.cache = {}

    def __call__(self, *args):
        if args in self.cache:
            return self.cache[args]
        else:
            value = self.func(*args)
            self.cache[args] = value
            return value
```

Decoradores

...Continuando con el ejemplo anterior:

```
@cached
def fibonacci(n):
    if n in (0, 1):
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Iteradores

Hemos visto anteriormente que en los ciclos `for` somos capaces de recorrer cierto tipo de objetos, como las listas o las cadenas. Esto es posible porque se trata de objetos **iterables**.

Iteradores

Hemos visto anteriormente que en los ciclos `for` somos capaces de recorrer cierto tipo de objetos, como las listas o las cadenas. Esto es posible porque se trata de objetos **iterables**.

Para que un objeto sea iterable, éste debe implementar el método `__iter__`, el cual retorna un tipo de objeto llamado **iterador**.

Iteradores

Hemos visto anteriormente que en los ciclos `for` somos capaces de recorrer cierto tipo de objetos, como las listas o las cadenas. Esto es posible porque se trata de objetos **iterables**.

Para que un objeto sea iterable, éste debe implementar el método `__iter__`, el cual retorna un tipo de objeto llamado **iterador**.

Un iterador es un objeto que cumple con dos características:

- Implementa el método `__iter__`, en el cual se retorna a sí mismo.
- Implementa el método `next`, el cual se encarga de retornar el próximo elemento de la colección cada vez que es invocado, o levanta una excepción del tipo StopIteration cuando ya no hay más elementos.

Iteradores

Cuando utilizamos un objeto secuencial en un `for`, en realidad el intérprete está obteniendo un iterador del objeto que queremos recorrer.

Iteradores

Cuando utilizamos un objeto secuencial en un `for`, en realidad el intérprete está obteniendo un iterador del objeto que queremos recorrer.

```
>>> a = [1, 2, 3, 4]
>>> it = a.__iter__()
>>> it
<listiterator object at 0x011E3DF0>
>>> for x in it:
...     print x
...
1
2
3
4
```

Iteradores

Ejemplo

```
import random

class RandomIter(object):
    def __init__(self, lst):
        self.lst = lst
        self.indexes = range(len(lst))
        random.shuffle(self.indexes)
        self.i = 0

    def __iter__(self):
        return self

    def next(self):
        if self.i < len(self.lst):
            self.i += 1
            return self.lst[self.indexes[self.i - 1]]
        else:
            raise StopIteration
```

Iteradores

```
class ListaRandom(list):
    def __iter__(self):
        return RandomIter(self)
```

Iteradores

```
class ListaRandom(list):
    def __iter__(self):
        return RandomIter(self)


>>> R = ListaRandom()
>>> R.extend(range(6))
>>> R
[0, 1, 2, 3, 4, 5]
>>> for x in R:
...     print x
...
4
2
5
3
0
1
```

Iteradores

Originalmente, son iterables las cadenas de texto y todas las colecciones nativas del lenguaje (listas, tuplas, conjuntos y diccionarios). Un número entero, por ejemplo, no es iterable.

Iteradores

Originalmente, son iterables las cadenas de texto y todas las colecciones nativas del lenguaje (listas, tuplas, conjuntos y diccionarios). Un número entero, por ejemplo, no es iterable.

```
>>> n = 50
>>> for i in n:
...     print i
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Iteradores

Ejercicio Práctico

Implementar un tipo de entero que sea iterable, y que vaya retornando cada vez el siguiente entero más cercano a cero, hasta llegar a cero.

Generadores

Un generador es un tipo de iterador especial, que se utiliza para generar ciertas secuencias. Para entender generadores es necesario entender primero la instrucción `yield`.

Generadores

Un generador es un tipo de iterador especial, que se utiliza para generar ciertas secuencias. Para entender generadores es necesario entender primero la instrucción `yield`.

```
>>> def uno_dos_tres():
...     yield 1
...     yield 2
...     yield 3
...
>>> uno_dos_tres()
<generator object uno_dos_tres at 0x014E5D78>
```

Generadores

Un generador es un tipo de iterador especial, que se utiliza para generar ciertas secuencias. Para entender generadores es necesario entender primero la instrucción `yield`.

```
>>> def uno_dos_tres():
...     yield 1
...     yield 2
...     yield 3
...
>>> uno_dos_tres()
<generator object uno_dos_tres at 0x014E5D78>
```

Ahora intentemos recorrer `uno_dos_tres()` con un ciclo for.
¿Qué sucede?...

Generadores

yield

La instrucción yield permite retornar progresivamente valores sin necesidad de ejecutar el código completo de la función que los genera. Esto es de bastante utilidad cuando queremos trabajar con secuencias muy grandes o incluso infinitas.

Generadores

yield

La instrucción yield permite retornar progresivamente valores sin necesidad de ejecutar el código completo de la función que los genera. Esto es de bastante utilidad cuando queremos trabajar con secuencias muy grandes o incluso infinitas.

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         yield a
...         a, b = b, a + b
...
>>> cont = 0
>>> for i in fib():
...     print i
...     if cont > 10:
...         break
...
```

Generadores

itertools

También podemos utilizar el módulo `itertools` para operar sobre iteradores y generadores:

```
>>> import itertools
>>> list(itertools.islice(fib(), 10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Generadores

itertools

También podemos utilizar el módulo `itertools` para operar sobre iteradores y generadores:

```
>>> import itertools
>>> list(itertools.islice(fib(), 10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

xrange()

Existe también `xrange`, que es una versión de `range` que funciona como un generador:

```
>>> for i in xrange(10):
...     print i
```

Muchas gracias!!