Python: Nivel 2

Contenido

Introducción

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. Luego de este curso el estudiante estará en plena capacidad para implementar sus propios proyectos con amplitud de poder aplicativo en el mundo real.

Funciones con número variable de argumentos

Hasta ahora sólo hemos definido funciones con una cantidad fija de parámetros, sin embargo, en Python es posible definir funciones sin conocer de antemano el número de argumentos que ésta puede recibir. Para esto se usan los parámetros `*args` y `**kwargs`

```
def prueba_args(param1, *args):
    print "param1 = %s" % param1
    for a in args:
        print "otro argumento = %s" % a
```

En este caso, viene en `args` una lista de valores sin ningún nombre asociado. Para recibir un diccionario con un número variable de argumentos con nombres, se utiliza `**kwargs`:

```
def prueba_kwargs(**kwargs):
    for k in kwargs:
        print "%s = %s" % (k, kwargs[k])
```

Modelo de ejecución: Todo es un objeto

Una de las particularidades de Python, es la idea de que *todo es un objeto*, y esto no significa que el usuario esté obligado a adoptar el paradigma de la Programación Orientada a Objetos al momento de programar.

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

En la primera clase utilizamos la función `dir()` para explorar los atributos del módulo `math`, ya esto nos da una idea de que un módulo es tratado internamente como un objeto. Si escribimos esto en el intérprete:

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

Nos daremos cuenta de que incluso un número tiene asociado una cantidad de atributos y métodos:

```
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__',
'__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__format__',
'__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__index__', '__init__',
'__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__', '__neg__',
'__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

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

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

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

Otra cosa que podemos probar, es que si invocamos la función `dir()` sin argumentos, esto nos devolverá una lista de todos los nombres definidos en el ámbito actual.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'n']
```

En `__builtins__` se encuentran definidas todas las funciones nativas y variables globales del lenguaje (como `dir`, `help`, etc).

Hagamos esta prueba:

```
>>> help(__builtins__.help)
```

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
```

Como podemos ver, la variable `n` es un nombre asociado al objeto `7`, y ambos se refieren al mismo objeto en memoria.

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
```

Para saber si dos nombres están asociados a un mismo objeto, se utiliza el operador `is`. Para preguntar si un valor es distinto de nulo, por ejemplo, lo correcto es evaluar la expresión `obj is not None`, en lugar de utilizar el operador de desigualdad (!=).

El siguiente experimento con listas nos explica un poco mejor la mecánica de los nombres y las asociaciones:

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

Adicionalmente, podemos probar que si modificamos el objeto asociado a `L1`, estaremos afectando directamente a `L2`.

```
>>> L1.append(5000)
>>> L2
[1, 2, 3, 5000]
```

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]
```

La función `map` ya viene implementada en el lenguaje Python, por ser una característica común de los lenguajes funcionales.

Es incluso factible asociar el nombre de una función a una nueva función o a algún otro valor. Para el intérprete sólo se trata de nombres y objetos:

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

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.

Sin embargo, podemos realizar la siguiente prueba:

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

La tupla `tup` no ha mutado en sí, ya que sigue constando de los mismos objetos (L, L). Aún así, podemos afectarla indirectamente modificando L, que por ser una lista, es un objeto mutable.

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 de estructuración.

Módulos

Los módulos son la abstracción organizativa más básica del lenguaje, y corresponden al código que se encuentra en un archivo con la extensión ".py" (esto no es requerido por el intérprete, pero es una convención altamente recomendable de seguir). El objetivo de un módulo es abarcar diversas funciones, clases y otros objetos conceptualmente relacionados, siguiendo 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 de por medio.

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, como ocurre con algunos lenguajes. Más bien, el código del módulo se mantiene aislado en su propio *espacio de nombres*, por lo que no es necesario preocuparnos de sobreescribir algún nombre que se encuentre definido en el archivo actual.

```
import math

def sqrt():
    print "hello world"

def raiz_cuadrada(n):
    return math.sqrt(n) # se especifica el espacio de nombres
```

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

```
from math import sqrt
```

Pero hay que tener cuidado, porque en este caso sí se pueden sobreescribir los nombres definidos en el ámbito actual.

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

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

Adicionalmente, Python permite hacer `from modulo import *`, lo cual importa todos los miembros de un módulo en el ambiente actual, pero esto es considerado una **muy** mala práctica.

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. Es posible importar un paquete completo como espacio de nombres, como si éste fuese un módulo:

```
import paquete

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

Al momento de hacer `import paquete.modulo`, primero se ejecuta el `__init__.py`, y posteriormente se ejecutan las definiciones de más alto nivel en `modulo.py`.

Es normal dejar en blanco el archivo `__init__.py` de un paquete, si no se necesitan definir nombres comunes.

Buenas prácticas

Renombramiento dinámico

Es recomendable evitar en la medida de lo posible el anidamiento excesivo de paquetes dentro de otros paquetes. Como está escrito en el Zen de Python: *"Es mejor plano que anidado"*. Sin embargo, algunos niveles de complejidad hacen inevitable el anidamiento de paquetes. En este caso, al momento de importar, podemos renombrar el módulo en cuestión, en beneficio de la legibilidad del código.

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

De esta forma evitamos escribir toda la cadena estructural al momento de acceder a dicho módulo.

Evitar la dependencia circular

Si tenemos en nuestro módulo `muebles.py` una clase *Mesa* y `Silla` que importan la clase `Carpintero` para responder una pregunta como `mesa.hecha_por()`, y simultáneamente la clase `Carpintero` necesita importar `Mesa` y `Silla` para satisfacer un método `carpintero.ha_fabricado()`, entonces tenemos una dependencia circular. Esto refleja un mal diseño de las entidades de nuestro proyecto, y en este caso tendremos que recurrir a artimañas como hacer "imports" dentro de métodos o funciones. Esta práctica debe evitarse.

Para evitar las dependencias circulares, debemos diseñar nuestro proyecto de forma jerárquica y en capas. Esto es, agrupar en un módulo las funcionalidades de más bajo nivel, y sobre esa capa diseñar otro módulo con un mayor nivel de abstracción, que utilice al módulo de bajo nivel pero no viceversa. Las dependencias en un proyecto deberían ir hacia un único sentido, esto no sólo evita las dependencias circulares, sino que hace el código mucho más mantenible (bajo "acoplamiento").

Evitar el código "Espagueti"

Se conoce como "código espagueti" a la práctica de programar con exceso de cláusulas anidadas y código redundante, así como llamadas de una función a otra sin un orden coherente o intuitivo. El problema de la dependencia circular también es una forma de "código espagueti".

En términos generales, si nos tomamos en serio las proposiciones escritas en El Zen de Python, no deberíamos terminar escribiendo un código complicado. En cualquier lenguaje es recomendable aprender a escribir de manera legible, pero en el caso de Python, la legibilidad es parte de la esencia del lenguaje.

PEP-8

Otra referencia sobre el estilo de programación que se requiere para programar en Python es un documento conocido como el PEP-8, en el cual están descritas todas las convenciones básicas de legibilidad y buenas prácticas.

Leer el PEP-8 es considerado casi obligatorio para los programadores que desean aprender la forma correcta de programar en Python.

En este documento no cubriremos el alcance del PEP-8, así que recomendamos al lector revisar por sí mismo la referencia.

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. Éste es un punto que vale la pena tomar en cuenta para desarrolladores que están acostumbrados a lenguajes como *Java*, por ejemplo.

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. Sin embargo, en la práctica, una clase probablemente heredará de otras clases, y definirá una serie de atributos.

Si una clase no hereda de alguna otra clase conocida, es recomendable definirla al menos como una subclase de `object`.

Herencia

Como es común en los lenguajes Orientados a Objetos, una clase puede "heredar" los atributos y métodos de otra clase. El nombre de la clase "padre" debe ser accesible en el ámbito en el que se define la clase hija. En caso de que la clase padre se encuentre definida en un módulo distinto, se utiliza la ruta completa como argumento:

```
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`, utilizando la siguiente sintaxis:

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

isinstance()

Para revisar que un objeto pertenezca a una clase dada, podemos utilizar la funciones `isinstance()`:

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

issubclass()

`issubclass()` se utiliza para saber si una clase hereda de otra:

```
>>> 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.

```
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.

Técnicamente, `__init__` no es un constructor, ya que la instancia ya se encuentra construida cuando este método se ejecuta. Además, no es obligatorio que una clase defina un método `__init__`. En caso

de definirse, éste se utiliza para inicializar todos los atributos necesarios de una instancia.

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

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`, el cual es una referencia a la instancia en cuestión. Adicionalmente, es posible que un método requiera de otros parámetros.

Las variables de instancia (o propiamente "atributos" como se les llama en otros lenguajes) no necesitan ser declarados, ya que una instancia de una clase se comporta dinámicamente. Es suficiente con inicializarlos desde el `__init__`, como en el ejemplo anterior.

Podemos comprobar que los objetos son dinámicos asignando a una instancia nuevos atributos en tiempo de ejecución:

```
>>> 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__`.

`__del__` es un método que se ejecuta cuando un objeto está apunto de eliminarse mediante una llamada a del().

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

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

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

Intentemos instanciar y destruir un objeto de la clase `Saludo`:

..code-block:: python

```
>>> s = Saludo("Bob")
>>> s.decir_hola()
'Hola Bob!'
>>> del s
'Adios!...'
```

Atributos de clase

Las variables declaradas explícitamente en una clase sin estar ligadas a `self` son consideradas atributos propios de la clase, aunque también son accesibles desde las instancias.

```
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
```

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
```

Podemos notar que en el código anterior se utiliza el atributo `__class__`, el cual es una referencia a la clase de la instancia actual. Las clases *también* son objetos.

Encapsulamiento

En Python no existe un mecanismo como tal para el encapsulamiento de atributos. Esto es posible mediante convenciones de nombramiento. Por lo general, cualquier nombre precedido por un guión bajo o *underscore* (_) se considera un atributo no público (o lo que en muchos lenguajes se conoce como atributos "protegidos"), y todo atributo precedido por doble guión bajo (__) es considerado privado. El correcto uso del encapsulamiento en Python, queda de parte de las prácticas concientes del programador.

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

De igual manera, no es una costumbre frecuente definir "getters" y "setters" por cada atributo de una clase. Es considerado completamente normal el acceso a los atributos directamente, en beneficio de la legibilidad.

Módulos nativos

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.

A continuación exploraremos algunos de los módulos nativos principales. La lista de módulos provistos por la biblioteca estándar de Python es extensa, para ver la lista y la documentación completa de los módulos se recomienda revisar este enlace.

datetime / time

El módulo `datetime` provee clases para manipular fechas y horas de diferentes maneras, incluyendo soporte para aritmética de fechas y horas, entre otras cosas.

```
>>> 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 días para mi cumpleaños" % tiempo_hasta_cumple.days
'Quedan 202 días para mi cumpleaños'
```

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))
```

El código anterior, elimina todos los archivos y subdirectorios desde el ámbito de `topdown`. Esto es sumamente peligroso!

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]}]'
```

random

En el módulo `random` están implementadas varias funcionalidades concernientes a generación de aleatoriedad.

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

Podemos incluso aleatorizar el orden de una lista:

```
>>> 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

El lenguaje Python provee un sintaxis simple pero bastante poderosa llamada "decoradores". 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. Esto es posible debido a que en Python las funciones son objetos de primer orden. La sintaxis de los decoradores la siguiente:

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

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

Este mecanismo es útil 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`.

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

Un ejemplo práctico puede ser implementar un decorador que funcione como *cache* de los resultados de una función. De esta manera, si una función se invoca más de una vez con los mismo argumentos, los resultados ya se encuentran guardados en memoria.

```
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

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

Iterables e 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. En el caso de no haber más próximos elementos, éste levanta una excepción del tipo `StopIteration`.

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

Consideremos una lista L, esta lista es un *iterable*, mas ella misma no es un iterador:

```
>>> a = [1, 2, 3, 4]
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x014E5D78>
>>> it = a.__iter__()
>>> it
<listiterator object at 0x011E3DF0>
```

Si quisiéramos implementar una clase que sea iterable, debemos implementar el protocolo descrito anteriormente:

```
import random

class MiLista(list):
    def __iter__(self):
        return MiRandomIter(self)

class MiRandomIter(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
```

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
```

Ejercicio práctico: implementar un tipo de entero que sea iterable, y que vaya retornando cada vez el siguiente entero más cercano 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`.

`yield` es una instrucción que puede utilizarse únicamente dentro de una función, y actúa de manera similar a `return`, sólo que en este caso la función retorna un *generador*, y conserva el estado de ejecución de dicha función.

Veamos el siguiente ejemplo:

```
>>> 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?...

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.

Podemos implementar, por ejemplo, una función que sea capaz de generar la secuencia fibonacci:

```
>>> 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
...
```

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]
```

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

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