Tipos de datos complejos

Python, posee además de los tipos ya vistos, 3 tipos más complejos, que admiten una **colección de datos**. Estos tipos son:

- Tuplas
- Listas
- Diccionarios

Estos tres tipos, pueden almacenar colecciones de datos de diversos tipos y se diferencian por su sintaxis y por la forma en la cual los datos pueden ser manipulados.

Tuplas

Una tupla **es una variable que permite almacenar varios datos inmutables** (no pueden ser modificados una vez creados) de tipos diferentes:

```
mi_tupla = ('cadena de texto', 15, 2.8, 'otro dato', 25)
```

Se puede acceder a cada uno de los datos mediante su índice correspondiente, siendo 0 (cero), el índice del primer elemento:

```
print mi_tupla[1]  # Salida: 15
```

También se puede acceder a una porción de la tupla, indicando (opcionalmente) desde el índice de inicio hasta el índice de fin:

```
print mi_tupla[1:4]  # Devuelve: (15, 2.8, 'otro dato')
print mi_tupla[3:]   # Devuelve: ('otro dato', 25)
print mi_tupla[:2]   # Devuelve: ('cadena de texto', 15)
```

Otra forma de acceder a la tupla de forma inversa (de atrás hacia adelante), es colocando un índice negativo:

```
print mi_tupla[-1]  # Salida: 25
print mi_tupla[-2]  # Salida: otro dato
```

Listas

Una lista es similar a una tupla con la diferencia fundamental de que permite modificar los datos una vez creados

```
mi_lista = ['cadena de texto', 15, 2.8, 'otro dato', 25]
```

A las listas se accede igual que a las tuplas, por su número de índice:

```
print mi_lista[1]    # Salida: 15
print mi_lista[1:4]  # Devuelve: [15, 2.8, 'otro dato']
print mi_lista[-2]   # Salida: otro dato
```

Las lista NO son inmutables: permiten modificar los datos una vez creados:

```
mi_lista[2] = 3.8  # el tercer elemento ahora es 3.8
```

Las listas, a diferencia de las tuplas, permiten agregar nuevos valores:

```
mi_lista.append('Nuevo Dato')
```

Diccionarios

Mientras que a las listas y tuplas se accede solo y únicamente por un número de índice, los diccionarios permiten utilizar una clave para declarar y acceder a un valor:

```
mi_diccionario = {'clave_1': valor_1, 'clave_2': valor_2, \
                  'clave_7': valor_7}
print mi_diccionario['clave_2']  # Salida: valor_2
```

Un diccionario permite eliminar cualquier entrada:

```
del(mi_diccionario['clave_2'])
```

Al igual que las listas, el diccionario permite modificar los valores

```
mi_diccionario['clave_1'] = 'Nuevo Valor'
```

Estructuras de Control de Flujo

Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada. En este capítulo, hablaremos sobre dos estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas

Identación

Para hablar de estructuras de control de flujo en Python, es imprescindible primero, hablar de identación.

¿Qué es la identación? En un lenguaje informático, la identación es lo que la sangría al lenguaje humano escrito (a nivel formal). Así como para el lenguaje formal, cuando uno redacta una carta, debe respetar ciertas sangrías, los lenguajes informáticos, requieren una identación.

No todos los lenguajes de programación, necesitan de una identación, aunque sí, se estila implementarla, a fin de otorgar mayor legibilidad al código fuente. Pero **en el caso de Python, la identación es obligatoria**, ya que de ella, dependerá su estructura.

> **PEP 8: identación**
> Una identación de **4 (cuatro) espacios en blanco**, indicará que las instrucciones identadas, forman parte de una misma estructura de control.

Una estructura de control, entonces, se define de la siguiente forma:

```
inicio de la estructura de control:
    expresiones
```

Encoding

El **encoding** (o codificación) es otro de los elementos del lenguaje que no puede omitirse a la hora de hablar de estructuras de control.

> El **encoding** no es más que una **directiva que se coloca al inicio de un archivo Python, a fin de indicar al sistema, la codificación de caracteres utilizada en el archivo.**

```
# -*- coding: utf-8 -*-
```

utf-8 podría ser cualquier codificación de caracteres. Si no se indica una codificación de caracteres, Python podría producir un error si encontrara caracteres “extraños”:

```
print "En el Ñágara encontré un Ñandú"
```

Producirá un error de sintaxis: `SyntaxError: Non-ASCII character[...]`

En cambio, indicando el encoding correspondiente, el archivo se ejecutará con éxito:

```
# -*- coding: utf-8 -*-

print "En el Ñágara encontré un Ñandú"
```

Produciendo la siguiente salida:

```
En el Ñágara encontré un Ñandú
```

Asignación múltiple

Otra de las ventajas que Python nos provee, es la de poder asignar en una sola instrucción, múltiples variables:

```
a, b, c = 'string', 15, True
```

En una sola instrucción, estamos declarando tres variables: a, b y c y asignándoles un valor concreto a cada una:

```
>>> print a
string
>>> print b
15
>>> print c
True
```

La asignación múltiple de variables, también puede darse utilizando como valores, el

contenido de una tupla:

```
>>> mi_tupla = ('hola mundo', 2011)
>>> texto, anio = mi_tupla
>>> print texto
hola mundo
>>> print anio
2011
```

O también, de una lista:

```
>>> mi_lista = ['Argentina', 'Buenos Aires']
>>> pais, provincia = mi_lista
>>> print pais
Argentina
>>> print provincia
Buenos Aires
```

Estructuras de control de flujo condicionales

> *"[...] Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición [...]"*
>
> *Cita textual del libro "Python para Todos" de Raúl González Duque (http://mundogeek.net/tutorial-python/)*

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. **La evaluación de condiciones**, solo **puede arrojar** 1 de 2 resultados: **verdadero o falso** (True o False).

En la vida diaria, actuamos de acuerdo a la evaluación de condiciones, de manera mucho más frecuente de lo que en realidad creemos: ***Si** el semáforo está en verde, cruzar la calle. **Sino**, esperar a que el semáforo se ponga en verde.* A veces, también evaluamos más de una condición para ejecutar una determinada acción: *Si llega la factura de la luz **y** tengo dinero, pagar la boleta.*

Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación):

OPERADORES RELACIONALES (DE COMPARACIÓN)

Símbolo	Significado	Ejemplo	Resultado
==	Igual que	5 == 7	Falso
!=	Distinto que	rojo != verde	Verdadero
<	Menor que	8 < 12	Verdadero
>	Mayor que	12 > 7	Falso
<=	Menor o igual que	12 <= 12	Verdadero
>=	Mayor o igual que	4 >= 5	Falso

Y para evaluar más de una condición simultáneamente, se utilizan **operadores lógicos**:

OPERADORES LÓGICOS

Operador	Ejemplo	Resultado*	
and (y)	5 == 7 and 7 < 12	0 y 0	Falso
	9 < 12 and 12 > 7	1 y 1	Verdadero
	9 < 12 and 12 > 15	1 y 0	Falso
or (o)	12 == 12 or 15 < 7	1 o 0	Verdadero
	7 > 5 or 9 < 12	1 o 1	Verdadero
xor	4 == 4 xor 9 > 3	1 o 1	Falso
(o excluyente)	4 == 4 xor 9 < 3	1 o 0	Verdadero

(*) 1 indica resultado verdadero de la condición, mientras que 0, indica falso.

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje: **`if`** (si), **`elif`** (sino, si) y **`else`** (sino).

Veamos algunos ejemplos:

Si semáforo esta en verde, cruzar la calle. Sino, esperar.

```
if semaforo == verde:
    print "Cruzar la calle"
else:
    print "Esperar"
```

Si gasto hasta $100, pago con dinero en efectivo. Sino, si gasto más de $100 pero menos de $300, pago con tarjeta de débito. Sino, pago con tarjeta de crédito.

```
if compra <= 100:
    print "Pago en efectivo"
elif compra > 100 and compra < 300:
    print "Pago con tarjeta de débito"
else:
    print "Pago con tarjeta de crédito"
```

Si la compra es mayor a $100, obtengo un descuento del 10%

```
importe_a_pagar = total_compra

if total_compra > 100:
    tasa_descuento = 10
    importe_descuento = total_compra * tasa_descuento / 100
    importe_a_pagar = total_compra – importe_descuento
```

Estructuras de control iterativas

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

En Python se dispone de dos estructuras cíclicas:

- El bucle **while**
- El bucle **for**

Las veremos en detalle a continuación.

Bucle while

Este bucle, se encarga de ejecutar una misma acción “mientras que” una determinada condición se cumpla:

> Mientras que año sea menor o igual a 2012, imprimir la frase “Informes del Año *año*”

```
# -*- coding: utf-8 -*-

anio = 2001
while anio <= 2012:
    print "Informes del Año", str(anio)
    anio += 1
```

La iteración anterior, generará la siguiente salida:

```
Informes del año 2001
Informes del año 2002
Informes del año 2003
Informes del año 2004
Informes del año 2005
Informes del año 2006
Informes del año 2007
Informes del año 2008
Informes del año 2009
Informes del año 2010
Informes del año 2011
Informes del año 2012
```

Si miras la última línea:

```
anio += 1
```

Podrás notar que en cada iteración, incrementamos el valor de la variable que condiciona el bucle (`anio`). Si no lo hiciéramos, esta variable siempre sería igual a `2001` y el bucle se ejecutaría de forma infinita, ya que la condición (`anio <= 2012`) siempre se estaría cumpliendo.

Pero ¿Qué sucede si el valor que condiciona la iteración no es numérico y no puede incrementarse? En ese caso, podremos utilizar una estructura de control condicional, anidada dentro del bucle, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada break:

```
while True:
    nombre = raw_input("Indique su nombre: ")
    if nombre:
        break
```

El bucle anterior, incluye un condicional anidado que verifica si la variable nombre es verdadera (solo será verdadera si el usuario tipea un texto en pantalla cuando el nombre le es solicitado). Si es verdadera, el bucle para (break). Sino, seguirá ejecutándose hasta que el usuario, ingrese un texto en pantalla.

Bucle for

El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

> Por cada nombre en mi_lista, imprimir nombre

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
    print nombre
```

> Por cada color en mi_tupla, imprimir color

```
mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')
for color in mi_tupla:
    print color
```

En los ejemplos anteriores, nombre y color, son dos variables declaradas en tiempo de ejecución (es decir, se declaran dinámicamente durante el bucle), asumiendo como valor, el de cada elemento de la lista (o tupla) en cada iteración.

Otra forma de iterar con el bucle for, puede emular a while:

> Por cada año en el rango 2001 a 2013, imprimir la frase "Informes del Año *año*"

```
# -*- coding: utf-8 -*-
for anio in range(2001, 2013):
    print "Informes del Año", str(anio)
```

03 Módulos, paquetes y namespaces

En Python, cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado `__init__.py`. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

Creando módulos empaquetados

En Python, cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado **`__init__.py`**. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

```
.
└── paquete
    ├── __init__.py
    ├── modulo1.py
    ├── modulo2.py
    └── modulo3.py
```

Los paquetes, a la vez, también pueden contener otros sub-paquetes:

```
.
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Y los módulos, no necesariamente, deben pertenecer a un paquete:

```
.
├── modulo1.py
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Importando módulos enteros

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario **importar los módulos** que se quieran utilizar. Para importar un módulo, se utiliza la instrucción **`import`**, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el .py) que se desee importar.

```
# -*- coding: utf-8 *-*

import modulo  # importar un módulo que no pertenece a un paquete
import paquete.modulo1  # importar un módulo que está dentro de un paquete
import paquete.subpaquete.modulo1
```

La instrucción `import` seguida de `nombre_del_paquete.nombre_del_modulo`, nos permitirá hacer uso de todo el código que dicho módulo contenga.

> **Python tiene sus propios módulos**, los cuales forman parte de su **librería de módulos estándar**, que también pueden ser importados.

Namespaces

Para **acceder** (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el ***namespace***, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra `import`, es decir la ruta (namespace) del módulo:

```
print modulo.CONSTANTE_1
print paquete.modulo1.CONSTANTE_1
print paquete.subpaquete.modulo1.CONSTANTE_1
```

Alias

Es posible también, abreviar los namespaces mediante un “alias”. Para ello, durante la importación, se asigna la palabra clave **`as`** seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

```
import modulo as m
import paquete.modulo1 as pm
import paquete.subpaquete.modulo1 as psm
```

Luego, para acceder a cualquier elemento de los módulos importados, el namespace utilizado será el alias indicado durante la importación:

```
print m.CONSTANTE _1
print pm.CONSTANTE _1
print psm.CONSTANTE_1
```

Importar módulos sin utilizar namespaces

En Python, es posible también, importar de un módulo solo los elementos que se desee utilizar. Para ello se utiliza la instrucción `from` seguida del namespace, más la instrucción `import` seguida del elemento que se desee importar:

```
from paquete.modulo1 import CONSTANTE_1
```

En este caso, se accederá directamente al elemento, sin recurrir a su namespace:

```
print CONSTANTE_1
```

Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco:

```
from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2
```

Pero ***¿qué sucede si los elementos importados desde módulos diferentes tienen los mismos nombres?*** En estos casos, habrá que **prevenir fallos, utilizando alias para los elementos**:

```
from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2
from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1, CONSTANTE_2 as CS2

print C1
print C2
print CS1
print CS2
```

> **PEP 8: importación**
> La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos.
> Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación.
> Entre cada bloque de imports, debe dejarse una línea en blanco.

De forma alternativa (pero muy poco recomendada), también es posible importar todos los elementos de un módulo, sin utilizar su namespace pero tampoco alias. Es decir, que todos los elementos importados se accederá con su nombre original:

```
from paquete.modulo1 import *

print CONSTANTE_1
print CONSTANTE_2
```

> **#TODO:** Abrir una terminal e iniciar el shell interactivo (intérprete) de Python. A continuación, importar el módulo `this`:
> **`import this`**