Clase 2: Tipos de datos y estructuras de control

Introducción

Una característica de Python es la cantidad y versátilidad de sus tipos de datos. Conocerlos en profundidad sirve para tener un buen criterio a la hora de elegir como modelar nuestros datos.

En líneas generales los tipos de datos se dividen en *primarios* y *derivados*:

- Primarios (o primitivos): No están basados en otro tipo de datos, como numericos (int, float, decimal, complex) y str (cadenas).
- Derivados: Agrupan a alguno de los anteriores, como listas, diccionarios, tuplas, etc. Se pueden subclasificar según distintos parámetros: Ordenados (o secuenciales) / Desordenados y Mutables / Inmutables.

Algunos ejemplos de datos primarios (enteros, flotantes, complejos y cadenas):

```
>>> type(5)
<type 'int'>
>>> type(5.0)
<type 'float'>
>>> type(5 + 5.0)
<type 'float'>
>>> 5 + 5.0
10.0
>>> type(2+3j)
<type 'complex'>
>>> (2+3j).real
2.0
>>> (2+3j).imag
3.0
>>> type('Hola!')
<type 'str'>
>>> 'hola' + ' mundo!'
'hola mundo!'
```

No pueden mezclarse valores númericos con cadenas:

```
>>> 'hela' + 2
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'hela' + 2
TypeError: cannot concatenate 'str' and 'int' objects
>>> 'hela' + str(2)
'hela2'
```

Este error se debe a que Python es un lenguaje de tipado dinámico pero fuerte. Es dinámico porque en cualquier momento se puede definir una variable con un tipo de dato distinto. En los lenguajes de tipado estático una vez que se crea un dato de un tipo particular, esto no puede alterarse. Python es de tipado fuerte porque una vez definido, este no puede convertirse automaticamente en otro tipo de datos. Por ejemplo en PHP '1' + 1 es igual a 2. En Python sumar un string con un int es imposible, porque Python no presupone nada con respecto a como hacer esta operación no definida. Lo que si es posible es "convertir" el '1' en 1 y obtener 2 (o convertir el 1 en '1' y obtener '11').

El "problema" de los números flotantes

El resultado de las operaciones con números flotantes puede ser inesperado:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

La causa de este problema es que Python no esconde el hecho que las operaciones que involucran numeros de punto flotante no son exactas debido a que hay un error inherente al pasar internamente los números de la base 2 (la que usa realmente el hardware) a la base 10. Este error ocurre con todos los lenguajes, la diferencia es que la mayoría de los lenguajes oculta este hecho usando algún tipo de redondeo. En Python este redondeo hay que hacerlo de manera explicita (si hace falta para la aplicación que estamos haciendo). Una manera de contrarrestar esto es con la función incorporada (built-in) *round()*. Esta función requiere 2 parámetros. El primero es el número que se quiere redondear y el segundo es la precisión con la que se quiere mostrar dicho valor. En el siguiente ejemplo se redondea el resultado de la suma a un solo decimal:

```
>>> round(0.1 + 0.1 + 0.1 - 0.3,1)
0.0
```

Alternativamente, para no perder precisión, existe el módulo **decimal**. A diferencia de las operaciones de punto flotante, con este módulo las operaciones se realizan directamente en base 10. Si bien el resultado en este caso es exacto, la operación es mas lenta debido a que se resuelve por software mientras que las de punt flotante aprovechan mejor el hardware.

Uso del módulo decima:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Mas información: http://docs.python.org/library/decimal.html y http://floating-point-gui.de

Cadenas (str o string)

Los string o cadenas son secuencias de símbolos ordenados. El manejo de cadenas en Python es bastante intuitivo. Hay que tener en cuenta que las cadenas son un tipo de datos inmutables. Una vez creadas no pueden modificarse. Veamos algunas operaciones comunes asociadas a las cadenas:

```
>>> 'Hola mundo!'
'Hola mundo!'
>>> a='Hola mundo!'
>>> len(a)
11
>>> a.lower()
'hola mundo!'
>>> a.lower()
'HOLA MUNDO!'
>>> a.count('o')
2
```

```
>>> a.center(14)
' Hola mundo!  '
```

El siguiente ejemplo muestra que las cadenas están indexadas comenzando desde el 0. También se muestra la diferencia entre las funciones *find()* e *index()*. Ambas retornan la posición en nuestra cadena de la cadena de búsqueda. La diferencia es que si la cadena de búsqueda no existe en nuestra cadena, *index()* retorna un error mientras que *find()* retorna "-1". Como esto último puede ser confuso, por lo que se prefiere usar *index()* antes que *find()*.

```
>>> a.find('H')
0
>>> a.find('mundo')
5
>>> a.find('e')
-1
>>> a.index('H')
0
>>> a.index('e')
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    a.index('e')
ValueError: substring not found
```

Las cadenas no se pueden modificar, lo que se puede hacer es crear otra cadena en base a la original. Por ejemplo para convertir una cadena de caracteres de mayúscula minúscula:

```
>>> a="Hola mundo!"
>>> a.lower()
'hola mundo!'
>>> a
'Hola mundo!'
>>> a = a.lower()
>>> a
'hola mundo!'
```

También se puede acceder a substrings usando la *notación de rebanadas* (*slice notation* in inglés) donde se usan indices que marcan las posiciones de inicio y fin de una porción. En lugar de indicar el número de elemento, se indica el espacio entre los elementos:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
```

Ejemplo de uso de *slice notation*:

```
>>> mi_str = "Python"
>>> mi_str[0:2]
'Py'
>>> mi_str[:2]
'Py'
>>> mi_str[4:6]
'on'
>>> mi_str[4:]
```

```
'on'
```

Otra función importante asociada a strings es *split()*, que devuelve una lista donde cada elemento de la lista es la parte de la cadena delimitada por el separador (en el caso del ejemplo de arriba, el separador es un espacio en blanco).

```
>>> a.split(' ')
['Hola', 'mundo!']
```

Para mas información sobre listas:

Listas

Las listas, al igual que las cadenas, son datos ordenados. Una diferencia importante es que es un tipo de dato derivado, también llamado *contenedor*. Las listas contienen como elementos a enteros, a cadenas o incluso a otras listas. Una lista es una secuencia ordenada de datos.

Las operaciones mas comunes son *append()* (agrega un elemento al final) y *pop()* (saca un elemento del final). A continuación algunas operaciones elementales con listas, como asi también la notación usada para acceder a sus elementos:

```
>>> mi_lista = [1,2,3]
>>> mi_lista.append(5)
>>> mi_lista
[1, 2, 3, 5]
>>> mi_lista.pop()
5
>>> mi_lista
[1, 2, 3]
>>> mi_lista + [4]
[1, 2, 3, 4]
>>> mi_lista
[1, 2, 3]
>>> mi_lista = mi_lista + [4]
>>> mi_lista
[1, 2, 3, 4]
>>> mi_lista.extend([5,6])
>>> mi_lista
[1, 2, 3, 4, 5, 6]
```

Al igual que los strings, las listas también están indexadas desde 0:

```
>>> mi_lista[0]
1
>>> mi_lista[3]
4
>>> mi_lista[3:5]
[4, 5]
>>> mi_lista[-2]
5
```

Y se puede utilizar la *slice notation*:

```
>>> mi_lista[3:5]
[4, 5]
```

Mas operaciones relacionadas a listas:

```
>>> variada = ['boga', 'cornalito', 'tararira']
>>> variada[2]
'tararira'
>>> variada[2][2:8]
'rarira'
>>> variada[2][2:]
'rarira'
>>> variada.append('pulpo')
>>> variada
['boga', 'cornalito', 'tararira', 'pulpo']
>>> variada.remove('cornalito')
>>> variada
['boga', 'tararira', 'pulpo']
>>> variada.sort()
>>> variada
['boga', 'pulpo', 'tararira']
>>> variada.index('pulpo')
1
>>> variada.index('pulpa')
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    variada.index('pulpa')
ValueError: list.index(x): x not in list
>>> 'pulpo' in variada
True
>>> 'pulpa' in variada
False
```

List comprenhesion

Como en teoria de conjuntos, las listas puede ser descriptas por extensión o por definición. En este último caso damos la regla para crearla sin especificar todos los elementos de manera taxativa:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Tuplas

Las tuplas son elementos ordenados como las listas, con la diferencia que son inmutables. Presten atención a los siguientes ejemplos, que tipo de métodos funcionan y cuales no:

```
>>> t1 = ('sgn1545',5,45)
>>> t1[0]
'sgn1545'
>>> 5 in t1
True
>>> t1.index(45)
2
>>> t1.count(45)
1
>>> t1.append(4)
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    t1.append(4)
AttributeError: 'tuple' object has no attribute 'append'
>>> t1.pop()
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    t1.pop()
AttributeError: 'tuple' object has no attribute 'pop'
>>> t1.remove(45)
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    t1.remove(45)
AttributeError: 'tuple' object has no attribute 'remove'
```

En este ejemplo se ve que están disponibles las funciones *count*, *index()* y el operador de pertenencia (*in*), pero los métodos para alterar las tuplas (*pop()* y *remove()*) no existen.

La decisión de usar una lista o un tupla a veces tiene mas que ver con la naturaleza de los datos que con las operaciones que querramos hacer con ellos. En principio las listas parecen mas útiles porque permiten ser modificadas a diferencia de las tuplas, por su parte las tuplas pueden ser índices de otros tipos de datos. Las listas deben ser usadas para datos del mismo tipo mientras que las tuplas se adecuan a datos de distinto tipo donde la posición indicaría el "campo".

La siguiente lista actuaría como una tabla de una base de datos, donde cada tupla corresponde a un registro (o una fila) y cada elemento de la tupla actúa como el contenido de un campo (o columna).

```
db = [("Jose","Paz",33),("Flor","Li",22)]
```

Diccionarios

Es una estructura contenedora y sin orden. Se lo usa para almacenar datos indexados por claves (pares claves/valor). Definición de nuevos diccionarios y propiedades:

```
>>> d = {'blue':'azul','red':'rojo'}
>>> d['blue']
'azul'
>>> d['azul']
Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    d['azul']
KeyError: 'azul'
```

Esto muestra que los diccionario son indexados por su clave y no por su valor. Una consecuencia de este hecho es que no puede haber claves repetidas.

Ahora veamos las propiedades mas usadas:

```
>>> 'blue' in d
True
>>> d.keys()
['blue', 'red']
>>> d.values()
['azul', 'rojo']
>>> d.items()
[('blue', 'azul'), ('red', 'rojo')]
>>> d.get('green','N/D')
'N/D'
>>> es2en = {}  #Diccionario vacio
>>> es2en['azul'] = 'blue'
>>> es2en
{'azul': 'blue'}
```

Diccionarios ordenados

Es una extensión de los diccionarios vistos anteriormente. Como su nombre lo indica, a diferencia de los diccionarios "clasicos", estos mantienen el orden interno de sus elementos. Para invocar este tipo de datos hay que importarlo desde el módulo **collections**:

```
>>> From collections import OrderedDict
>>> Markers = OrderedDict()
>>> Markers ['ms123'] = 23.9
>>> Markers ['mk31'] = 12.8
>>> Markers ['ms92'] = 32.1
```

Si no hace falta mantener el orden, es preferible usar el diccionario estándar ya que es mas rápido para ciertas operaciones.

Sets (conjuntos)

Al igual que los diccionarios los sets son contenedores sin orden. La característica principal es que sus elementos son únicos y permiten operaciones asociadas a conjuntos.

Una manera de crear un diccionario es creando uno vacio y luego agregarle los elementos. Observá que se usa *add* en lugar de *append* (cono en el caso de las listas).

Creación de conjuntos, agregado y remoción de elementos y operaciones básicas:

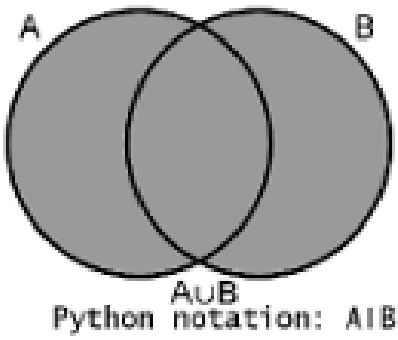
```
>>> mi_set = set()
>>> mi_set.add('juan')
>>> mi_set.add('nati')
>>> mi_set.add('viki')
>>> mi_set
set(['juan', 'viki', 'nati'])
>>> mi_set.pop()
'nati'
>>> mi_set
set(['juan', 'viki'])
>>> mi_set.add('nati')
>>> mi_set
set(['juan', 'viki', 'nati'])
```

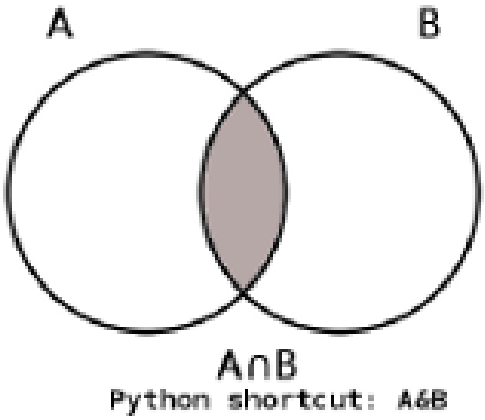
```
>>> otro = set(['juan','karina','diana'])
>>> otro
set(['diana', 'juan', 'kari'])
```

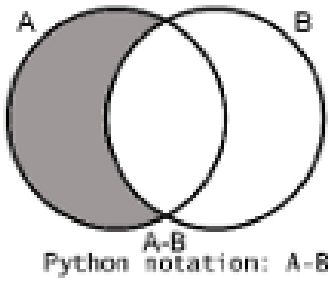
Veamos las operaciones típicas de los conjuntos:

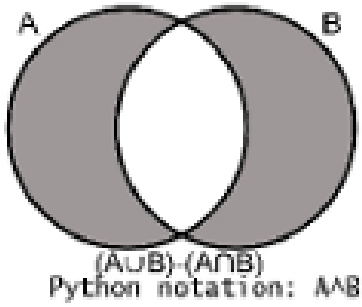
```
>>> mi_set.intersection(otro)
set(['juan'])
>>> mi_set.union(otro)
set(['viki', 'nati', 'diana', 'juan', 'kari'])
>>> mi_set.difference(otro)
set(['viki', 'nati'])
```

Representaciones de algunas operaciones de conjuntos:









Referencias tipos de datos

Estructuras de datos

- http://docs.python.org/tutorial/datastructures.html

Strings

- http://docs.python.org/library/string.html
- http://www.network-theory.co.uk/docs/pytut/Strings.html

Listas

- http://effbot.org/zone/python-list.htm

Diccionarios

- http://diveintopython.org/getting_to_know_python/dictionaries.html
- http://www.python.org/dev/peps/pep-0372/

Sets

- PEP-218 http://www.python.org/dev/peps/pep-0218/
- http://www.devshed.com/c/a/Python/Python-Sets/
- http://en.wikibooks.org/wiki/Python_Programming/Sets

Estructuras de control

Python tiene tres estructuras de control. Una de decisión (*if*) y dos de ciclos (*for* y *while*).

if

Modo general

```
if <expresion1>:
  <Instrucciones1>
elif <expresion2>:
  <Instrucciones2>
else:
  <Instrucciones3>
```

En caso que <*expresion1*> sea verdadera, solo se ejecuta el bloque de código llamado <*Instrucciones1*>, si <*expresion1*> es falsa, se evalua <*expresion2*>, que en caso de ser cierta, se ejecutan el bloque de código designado <*Instrucciones2*>. Puede haber tantos *elif* como sea necesario. En caso que ninguna expresión sea verdadera, el flujo del programa entra en el bloque del *else*, esto es en este esquema, <*Instrucciones3*>.

Este es el momento de notar que cada bloque de código está delimitado por los espacios agregados adelante de las instrucciones. Esto se lo llama *indentación* (o identación o sangría, según a quien le pregunten). Normalmente se usan 4 espacios por cada "nivel de indentación". También se pueden usar *tabs*, lo importante es no mezclar tabs con espacios. En otros lenguajes se utilizan llaves ({}) para delimitar los bloques de código y la indentación es opcional. En Python no se utilizan las llaves y la indentación es obligatoria.

Nota: Si bien se usan 4 espacios, en este libro usamos 2 espacios para la indentación porque los lectores de libros electrónicos (ebook readers) suelen tener un ancho muy limitado.

Ejemplo de uso

```
if puntos == 10:
  print("Felicitaciones!")
elif 5 <= puntos < 10:
  print("Muy bien")
else:
  print("Mal")
```

Otro ejemplo.

Si el objeto llamado *coord* es distinto a la cadena 'N/A', llamar *year* a los últimos 4 elementos del primer elemento de la lista *coord*:

```
if coord != 'N/A':
  year = int(coord[0][-4:])
```

for

for es usado para "recorrer" un objeto iterable.

Modo general

```
for <var> in <iterable>:
  <instrucciones>
[else:
  <instrucciones>]
```

Ejemplo de uso

El siguiente código sirve para recorrer la lista [1,3,4], y en cada iteración *x* toma el valor de cada elemento de la lista.

```
for x in [1, 3, 4]:
  print x
```

El bloque correspondiente al *else* en el *for*, que es optativo y casi nunca se usa, se ejecuta cuando el bloque bajo *for* se consume en su totalidad sin interrupciones producidas por *break* (ver mas adelante).

> ***¿Qué es un iterable?***
>
> Un iterable puede ser: Lista, string, tupla, diccionario, set o cualquier objeto que tenga el método *next()*.

while

Es para ejecutar un bloque de código repetidas veces, siempre y cuando <*expresion*> sea verdadera. Normalmente dentro del bloque de código hay alguna instrucción que cambia un valor para que esto ocurra y el ciclo termine. Otra alternativa para terminar el ciclo es usando **break** (ver mas adelante).

Modo general

```
while <expresion>:
  <instrucciones>
```

Ejemplo de uso

Imprimir elementos de *mi_set* hasta que este objeto contenedor (el set) se agote (y por lo tanto sea evaluado como falso):

```
mi_set = set([1,2,3,4])
while mi_set:
  print mi_set.pop()
```

Break

Break se usa para salir del ciclo que lo contiene. No confundir con saltos incondicionales de otros lenguajes. Ejemplo de uso de **break**:

```
for data in datasource:
  if data * 20 > 450:
    #hacer algo una sola vez
    break
```

La utilidad de **break** es que permite que hagamos algo una vez dentro de un ciclo y no tengamos que seguir recorriendolo en su totalidad (a menos que justo sea accedido en la última pasada).