Python Programming

Módulo 1

Tipos de datos y Colecciones

Tipos de datos

Python es un lenguaje de “tipado” dinámico. Esto quiere decir que las variables, como se las conoce en otros lenguajes, no son una suerte de cajas que aceptan únicamente un tipo de datos, sino más bien etiquetas que se le asignan a un objeto de cualquier tipo. No es necesario ahondar en cómo el intérprete maneja internamente las variables, simplemente tener en cuenta que no están ligadas a un tipo de dato en específico, y que acostumbramos a llamarlas *“objetos”* (ya veremos por qué).

Python tiene los tipos de dato *entero(int), flotante(float), complejo(complex), booleanos(bool)* y *strings(str).*

Podemos asignar el nombre que queramos, respetando no usar las palabras reservadas (lista de palabras reservadas) de Python ni espacios, guiones o números al principio.

Tipos de datos

Comencemos por abrir la consola interactiva de Python y escribir lo siguiente.

(A partir de ahora, siempre que veas los caracteres ">>>" da por sentado que estamos trabajando sobre la consola interactiva).

```
>>> dato = 1
```

Hemos creado un objeto que contiene el número 1. Escribiendo el nombre de un objeto en la consola interactiva obtenemos su valor.

```
>>> dato
1
```

La función type(), nos permite averiguar el tipo de dato de una variable.

```
>>> type(dato)
<class 'int'>
```

Como los objetos no están ligados a un tipo específico de datos, ahora podemos reemplazar el contenido de `dato` por una cadena:

```
>>> dato = "Hola mundo"
```

(En Python las cadenas se construyen con comillas dobles o simples).

Tipos de datos

Ya hemos visto dos tipos de datos: un número entero (`int`) y una cadena (`str`). Continuemos creando un número de coma flotante y un booleano.

```
>>> b = True
```

Un objeto booleano puede contener los valores `True` o `False` (ambos valores con Mayúscula la primera letra).

Los valores flotantes la coma como punto.

```
>>> pi = 3.14
```

Y por último no tan utilizados los complejos:

```
>>> z = 5 + 3j
```

(Este ejemplo tiene parte real con valor 5 y parte imaginaria 3)

None

Cuando queremos crear un objeto, pero por el momento no asignarle ningún valor, generalmente se utiliza `None` (en inglés, literalmente, *"nada"*). Esta palabra se usa para que la variable (u objeto) no tenga un tipo de dato asociado. None, no es un tipo de dato.

```
>>> a = None
>>> type(a)
<class 'NoneType'>
```

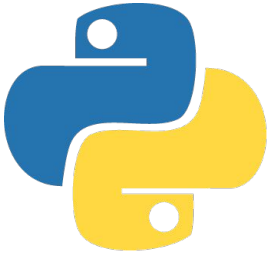
Casting

Cast o *Casting* significa convertir un tipo de dato a otro. Vimos tipos de datos como los `int`, `string` o `float`. Pues bien, es posible convertir de un tipo a otro.

- **Conversión implícita:** Es realizada automáticamente por Python. Sucede cuando realizamos ciertas operaciones con dos tipos distintos.

```
>>> a = 1
>>> b = 2.5
>>> a = a + b
>>> print(a)
>>> 3.5
```

Podemos ver como internamente Python ha convertido el `int` en `float` para poder realizar la operación, y la variable resultante es `float`.



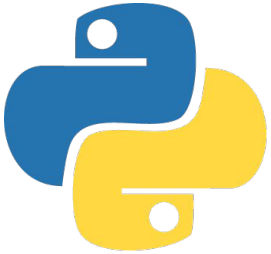
Casting

Conversión explícita: Es realizada expresamente por nosotros, como por ejemplo convertir de `str a int` con `str()`.

```
>>> a = 10.4
>>> int(a)
>>> 10

>>> b = 50
>>> str(b)
>>> '50'
>>> edad = "45"
>>> int(edad)
>>> 45
```

Podemos hacer conversiones entre tipos de manera explícita haciendo uso de diferentes funciones que nos proporciona Python. Las más usadas son las siguientes: `float(), str(), int()`



Colecciones

Python tiene una variedad de colecciones para agrupar datos en estructuras de diferentes tipos. Tenemos colecciones ordenadas o no ordenadas y colecciones mutables o inmutables.

Las c*olecciones ordenadas* son aquellas que pueden ser indexadas, es decir, que se pueden usar números enteros para acceder a sus elementos. Son colecciones ordenadas (o secuencias) las listas y las tuplas. Como colecciones no ordenadas tenemos los diccionarios. También existen los sets y los frozensets, estos últimos no tan empleados.

Listas

Las listas son objetos mutables (es decir, su contenido puede variar) y ordenados. En otros lenguajes existen estructuras similares. Podemos acceder a sus elementos a partir de su posición, indicando un índice entre corchetes, comenzando desde el 0.

```
>>> datos = [1, 2, 3]
>>> datos[0]
1
>>> datos[1]
2
>>> datos[2]
3
```

Los índices pueden ser negativos, para indicar que se debe empezar a contar desde el último elemento:

```
>>> datos[-1]
3
>>> datos[-2]
2
```

Listas

La indexación se puede realizar con uno, dos o tres parámetros. Con dos parámetros se puede seleccionar porciones (slices) de la secuencia y el tercer parámetro indica el paso con el que los elementos deben tomarse.

```
>>> animales =
["Gato","Perro","Gallo","Caballo","Tiburon","Pajaro"]
>>> animales[2:5]
['Gallo', 'Caballo', 'Tiburon']

>>> animales[0:5:2]
['Gato', 'Gallo', 'Tiburon']
```

Listas

Los elementos dentro de una lista no necesariamente tienen que ser del mismo tipo. Incluso pueden contener otras listas.

```
>>> b = [3.14, True, ["Hola mundo", False]]
>>> b[0]
3.14
>>> b[1]
True
>>> b[2]
['Hola mundo', False]
>>> b[2][0]
'Hola mundo'
>>> b[2][1]
False
```

Como las listas son objetos mutables, podemos cambiar el objeto en una posición determinada con una simple asignación.

```
>>> a = [1, 2, 3, 4]
>>> a[2] = "Hola mundo"
>>> a
[1, 2, 'Hola mundo', 4]
```

Listas

Las listas poseen funciones propias, métodos, con los cuales podemos trabajarlas de forma más óptima.

Para añadir un elemento al final de una lista, utilizamos el `append()`.

```
>>> a = [1, 2, 3, 4]
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
```

Y para insertar un elemento en una posición específica, la función `insert()` toma como primer argumento un índice de base 0 seguido del objeto a insertar.

```
>>> a.insert(2, -1)
>>> a
[ 1, 2, -1, 3, 4, 5]
```

Existen otros métodos que poseen las listas, para más información podemos ver la documentación oficial que habla sobre el tema: más sobre listas.

Tuplas

Las tuplas son similares a las listas, pero son objetos inmutables. Una vez creada una tupla, no pueden añadirse ni removerse elementos.

```
>>> t = (1, 2, 3, 4)
```

Se crean indicando sus elementos entre paréntesis (aunque también pueden omitirse) y, al igual que las listas, pueden contener objetos de distintos tipos, incluso otras tuplas. Se accede a sus elementos indicando su posición entre corchetes.

Es importante aclarar que, para crear una tupla con un único elemento, se debe usar (nótese la coma):

```
>>> x = (1,)
```

Ya que los paréntesis son también utilizados para agrupar expresiones y, de lo contrario, Python no podría distinguir si se trata de una expresión o de una tupla con un único elemento.

Siempre que requieras de un conjunto ordenado de objetos, pensá si su contenido será modificado en el transcurso del programa. En caso afirmativo, creá una lista. De lo contrario, una tupla.

Diccionarios

El último tipo de colección que veremos se llama *diccionario.* Es lo que en otros lenguajes se conoce como *vector* o *lista asociativa.* Está constituido por pares de una clave y un valor.

```
>>> d = {"a": 1, "b": 2}
```

En este caso, las claves son las cadenas **“a”** y **“b”**, que están asociadas a los números 1 y 2, respectivamente. Así, accedemos a los valores a partir de su respectiva clave.

```
>>> d["a"]
1
>>> d["b"]
2
```

Los valores en un diccionario pueden ser de cualquier tipo, también otros diccionarios. Las claves solo pueden ser objetos inmutables (por ejemplo, cadenas, enteros, números de coma flotante, tuplas; no así listas) y no pueden repetirse dentro de un mismo diccionario.

```
>>> d = {123: "Hola, mundo", True: 3.14,
(1, 2): False}
>>> d[123]
'Hola, mundo'
>>> d[True]
3.14
>>> d[(1, 2)]
False
```

Diccionarios

Los pares clave-valor de un diccionario no pueden ser accedidos a partir de un índice porque no es una colección ordenada.

Para cambiar el valor de una clave mantenemos la misma sintaxis que en las listas:

```
>>> d = {"a": 1, "b": 2}
>>> d["b"] = 3.14
>>> d
{'a': 1, 'b': 3.14}
```

Lo cual también sirve para añadir elementos:

```
>>> d["c"] = "Hola mundo"
>>> d
{'a': 1, 'c': 'Hola mundo', 'b': 3.14}
```

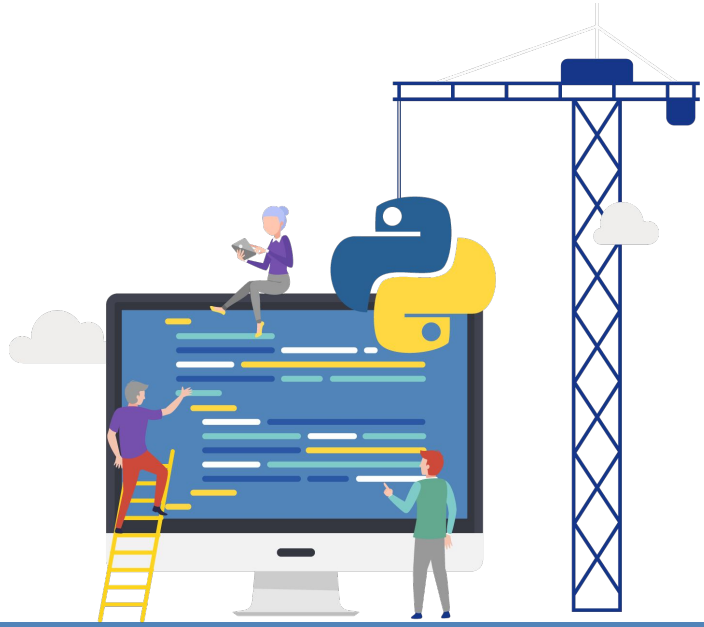
Y para remover un par clave-valor:

```
>>> del d["b"]
>>> d
{'a': 1, 'c': 'Hola mundo'}
```

len( ) y del

Para conocer la cantidad de elementos de una colección indistintamente del tipo, utilizamos la función `len()`. El `len()` es una función propia del lenguaje, NO es un método puntual de alguna colección.

```
>>> len([1, 2, 3])
3
>>> len((True, False))
2
>>> len({"a": 1})
1
>>> len("Hola mundo")
10
```



len( ) y del

La palabra clave **“del”** en Python se usa principalmente para eliminar objetos. Dado que todo en Python representa un objeto de una forma u otra. **“del”** se puede usar para eliminar una lista, eliminar una parte de una lista, eliminar diccionarios, eliminar pares clave-valor de un diccionario, eliminar variables, etc.

```
>>> numeros = [10,20,30]
>>> del numeros[1]
>>> numeros
>>> [10, 30]

>>> x = 10
>>> x
>>> 10
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Unpacking

Las listas y tuplas soportan un método llamado *unpacking* (cuya traducción podría ser *"desempaquetamiento"*). Veámos en un ejemplo:

```
>>> t = (1, 2, 3)
>>> a, b, c = t
>>> a
1
>>> b
2
>>> c
3
```

La operación `a, b, c = t` equivale a:

```
>>> a = t[0]
>>> b = t[1]
>>> c = t[2]
```

Es importante que la cantidad de objetos a la izquierda coincida con el número de elementos en la colección a la derecha.

¡Muchas gracias!

¡Sigamos trabajando!