clase1

November 22, 2023

1 Curso de Python - clase 1

1.1 Tipos de datos en Python

Los datos en Python pueden tener la forma de:

Tipos primitivos

Tipos no primitivos

Los tipos primitivos es la forma mínima que puede tener un dato. Sobre estos tipos se pueden crear estructuras de datos más complejas. Los primitivos de Python son:

Integer: numeros enteros

Float: numeros decimales (o con punto flotante)

String: cadenas de caracteres (texto, basicamente)

Boolean: variable binaria (sí o no, verdadero o falso, prendido o apagado)

A partir de estos tipos primitivos, la información se puede estructurar de formas más complejas pero muy útiles:

List: conjunto indexado y desordenado de variables de cualquier tipo (primitivo o no)

Tuple: son como las listas pero inmutables, es decir que una vez que creamos una tupla no podemos eliminar ni reemplazar sus elementos por otros.

Set: son como las listas pero todos sus elementos son del mismo tipo, el orden de los elementos es aleatorio y no deja guardar elementos repetidos. A veces es util convertir una lista en set cuando queremos rápidamente eliminar los repetidos. Son más eficientes para guardar info.

Dictionary: los diccionarios guardan datos en la forma de claves asociadas a un valor. Eso nos permite acceder a datos a través de un nombre en vez de un índice numérico.

File: Python base permite trabajar con archivos de nuestro sistema operativo. Obviamente es más fácil trabajar con archivos de texto (leerlos, escribirlos, eliminar cosas) que leer archivos binarios de otros programas (como excel). Para leer binarios hay muchas librerias que nos ayudan a trabajar con ellos.

Objetos: Si ninguna de estas estructuras de datos te gusta, podés crear una. Hasta podés decirle a un objeto que tipo de acciones es capaz de hacer. Dejmoslo para otra ocación…

Lectura recomendada: Tipos de datos en DataCamp

1.1.1 ¿Cómo se ven los primitivos?

[66]:
```
# Númericos
# La función type es una manera de preguntarle a Python el tipo de una variable
entero = -5
print(type(entero))
```

```
<class 'int'>
```

[67]:
```
flotante = 3.14
print(type(flotante))
```

```
<class 'float'>
```

[68]:
```
cadena = "Soy una cadena de caracteres"
print(type(cadena))
```

```
<class 'str'>
```

[69]:
```
cadena_larga = """
Esta es una manera de escribir cadenas de caracteres super largas.
Es útil cuando queremos escribir cosas tengan muchas líneas de extención.
También podemos usar comillas simples en los dos casos, son intercambiables.
"""
print(type(cadena_larga))
```

```
<class 'str'>
```

[70]:
```
booleano = True
booleano = False

print(type(booleano))
```

```
<class 'bool'>
```

1.1.2 Ejemplos de no primitivos más usados

[71]:
```
lista = [entero, flotante, cadena, booleano]
print(lista)
```

```
[-5, 3.14, 'Soy una cadena de caracteres', False]
```

[72]:
```
print(type(lista))
```

```
<class 'list'>
```

[73]:
```
tupla = (entero, flotante, cadena, booleano) # los paréntesis son opcionales
print(type(tupla))
```

```
<class 'tuple'>
```

[74]:
```
_set = {"hola", "esto","es","un","set","set","set"}
print(_set)
```

```
{'set', 'un', 'es', 'hola', 'esto'}
```

[75]:
```
print(type(_set))
```

```
<class 'set'>
```

[76]:
```
diccionario = {"uno": 1, "lista": lista, "otro_dict":{"vieron_inception?":
 ↪True}, "texto": "mamushka_vale"}
print(diccionario)
```

```
{'uno': 1, 'lista': [-5, 3.14, 'Soy una cadena de caracteres', False],
'otro_dict': {'vieron_inception?': True}, 'texto': 'mamushka_vale'}
```

[77]:
```
print(type(diccionario))
```

```
<class 'dict'>
```

1.1.3 Conversiones de tipos. (en inglés el verbo sería *to cast*)

los tipos en Python tienen sus palabras reservadas y permiten pasar de un tipo a otro siempre que se pueda. Si no se puede, la operación dará error

[78]:
```
# Solo unos ejemplos, vayan explorando.
# Cosas que se pueden
nuevo_int=int(4.5) # en este caso lo trunca
# Cosas que no
nuevo_dict = dict(lista)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/luko/indec/curso-python-indec/clase1.ipynb Celda 18 line 5

      <a href='vscode-notebook-cell:/home/luko/indec/curso-python-indec/clase1.
 ↪ipynb#X23sZmlsZQ%3D%3D?line=2'>3</a> nuevo_int=int(4.5) # en este caso lo␣
 ↪trunca
      <a href='vscode-notebook-cell:/home/luko/indec/curso-python-indec/clase1.
 ↪ipynb#X23sZmlsZQ%3D%3D?line=3'>4</a> # Cosas que no
----> <a href='vscode-notebook-cell:/home/luko/indec/curso-python-indec/clase1.
 ↪ipynb#X23sZmlsZQ%3D%3D?line=4'>5</a> nuevo_dict = dict(lista)

TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

1.2 Operadores y expresiones

Una expresión es cualquier parte de código de python que al ejecutarse de por resultado un valor. Para ello las expresiones hacen uso de símbolos especiales que llamamos operadores. Les dejo una ayuda de los operadores más usados, lógicos y aritméticos.

1.2.1 Operadores aritméticos

OPERADOR	DESCRIPCIÓN	USO
and	Devuelve True si ambos operandos son True	a and b
or	Devuelve True si alguno de los operandos es True	a or b
not	Devuelve True si alguno de los operandos False	not a

1.2.2 Operadores relacionales

OPERADOR	DESCRIPCIÓN	USO
and	Devuelve True si ambos operandos son True	a and b
or	Devuelve True si alguno de los operandos es True	a or b
not	Devuelve True si alguno de los operandos False	not a

1.2.3 Operadores lógicos

OPERADOR	DESCRIPCIÓN	USO
and	Devuelve True si ambos operandos son True	a and b
or	Devuelve True si alguno de los operandos es True	a or b
not	Devuelve True si alguno de los operandos False	not a

Lectura recomendada: Free code camp en español

1.3 Estructuras de control de flujo

En cualquier típico lenguaje de programación debemos poder especificar como se va a ejecutar nuestro código. No siempre queremos que se ejecuten todas las sentencias que escribimos. A veces vamos a querer que algunos procesos se ejecuten condicionalmente, otras veces vamos a querer repetir muchas veces el mismo proceso.

Estas estructuras tienen la forma de una sentencia con una palabra reservada (if, for, while, etc) y un cuerpo que corresponde a las lineas a ejecutar dentro de ese bloque de control. El cuerpo de estas estructuras debe estar indentado, con `tab` o `espacio` (en general solemos usar cuatro espacios), tratando de ser consistente en la indentación.

Lectura recomendada: Sentencias de control de flujo

1.3.1 Condicionales

La palabra reservada es if, la condición debe ser una expresión equivalente a un `boolean`, es decir True o False. Si queremos que se ejecute algo en caso de no cumplirse la condición podemos escribir `else`. Si queremos que se revisen en serie varias condiciones podemos usar la palabra reservada `elif`.

Veamos un ejemplo:

[ ]:
```
# Si A es igual a B
A="foo"
B="baz"

if A == B:
    print("son iguales")
else:
    print("No son iguales")
```

```
No son iguales
```

[ ]:
```
C="foo"

if A == B:
    print("A es igual a B")
elif A == C:
    print("A es igual a C")
else:
    print("no hay nada igual a A")
```

```
A es igual a C
```

[ ]:
```
id(A) == id(B)
```

[ ]:
```
False
```

NOTA: para estas comparaciones podriamos usar la palabra reservada `is` en lugar de `==` pero es recomendable usar `==` si quieren asegurarse que lo que se está comparando es el valor de esas variables. `is` compara si es la misma variable; o, dicho de modo brusco, compara si es el mismo cacho de memoria. En este caso, como Python no es tonto, no va a gastar memoria para guardar exactamente lo mismo en A y B.

En general va a funcionar perfectamente, pero cuando esten usando estructuras de datos mas complejas puede que se comporte distinto a lo que esperan.

1.3.2 Bucles for y while

A veces tenemos que repetir el mismo proceso múltiples veces. Para eso vamos usar estos bucles.

Bucle for En Python tenemos estas cosas a las que le llamamos iterables, básicamente son tipos complejos que pueden tener un índice que permita acceder a muchas variables distintas. Hay unos iterables más raros que no vamos a tocar en este curso. ¡Pero hay otros iterables que ya vimos!

Las listas, tuplas, sets y diccionarios SON iterables. Si queremos hacer una operación por cada elemento de estos, vamos a usar un bucle for. Veamos un ejemplo de lista y otro de diccionario:

[85]:
```
lista_de_enteros = [10,20,30,40,50]

for numero in lista_de_enteros:
    print(numero + 1)
```

```
11
21
31
41
51
```

[80]:
```
# ¿Y como hacemos para modificar la lista?

# Paso 1:
```

```
# Podemos acceder a los elementos de la lista por su índice (EMPIEZA POR 0 COMO␣
 ↪EL DIOS DE LA PROGRAMACIÓN MANDA, no como los degenerados de R que arrancan␣
 ↪por 1...)

lista_de_enteros[3]
```

[80]: 40

[81]:
```
# Ah pero cuando recorra la lista tengo que modificar cada elemento, pero para␣
 ↪ello tengo que saber su índice al momento de recorrerla
# Paso 2: enumerate al rescate!
# enumarate es una función de Python base que toma un
# iterable y se convierte en otro iterable solo que ahora sus elementos son␣
 ↪tuplas compuestas por el elemento original más su índice
for (indice,  numero) in enumerate(lista_de_enteros):
    print(f"Índice: {indice}, Numero originial: {numero}")
```

```
Índice: 0, Numero originial: 10
Índice: 1, Numero originial: 20
Índice: 2, Numero originial: 30
Índice: 3, Numero originial: 40
Índice: 4, Numero originial: 50
```

[86]:
```
# Paso 3: asignar los nuevos numeros a la lista original

for (indice,  numero) in enumerate(lista_de_enteros):
    lista_de_enteros[indice] = numero + 1

print(lista_de_enteros)
```

```
[11, 21, 31, 41, 51]
```

[87]:
```
# Si quieren poner esos valores en una nueva lista pueden crear una nueva y␣
 ↪usar el método append (como lo sugiere la palabra en inglés, solo le agrega␣
 ↪nuevas cosas al final)
nueva_lista = [] # Es una lista vacia!
for numero in lista_de_enteros:
    nueva_lista.append(numero - 5)

nueva_lista
```

[87]: [6, 16, 26, 36, 46]

[88]:
```
# Pro tip: Hay una forma que al principio es media confuza para hacer esto␣
 ↪escribiendo menos, se las dejo para que sepan que existe pero no le presten␣
 ↪atención
# en inglés se llama comprehension. En este caso es list comprehension.
```

```
lista_de_enteros = [numero/10 for numero in lista_de_enteros]

print(lista_de_enteros)
```

```
[1.1, 2.1, 3.1, 4.1, 5.1]
```

1.3.3 Bucle while

A veces no tenemos un iterable como los anteriores pero aún así queremos repetir un proceso hasta que determinada condición sea True. Para eso usamos el bucle while

[94]:
```
# Contemos hasta 100 con while
a = 0
while a != 100:
    a+=1

print(a)
```

```
100
```

¿Qué pasa si queremos escapar de los bucles por más que no se cumpla la condición? Usamos la palabra reservada `break`.

[95]:
```
while True: # Este bucle es infinito
    a-=1 # vamos a ir restandole 1 al valor de a que ahora quedo en 100
    if a == 11: # cuando a sea igual a 11, salimos del bucle
        break

print(a)
```

```
11
```

Bueno, yo se que deben estar muertos de cansancio ya ,pero haganme el aguante con una última cosa. ### ¡Funciones!

En el mundo de la programación tratamos de seguir una regla a la que llamamos DRY (don't repeat yourself), es decir, "no te repitas". En este mundo, la regla del menor esfuerzo es nuestra mayor virtud, cuanto menos código escribamos para resolver una tarea, menos probabilidades de introducir errores vamos a generar.

Para eso usamos las funciones, encapsulamos fragmentos de código reutilizable para solo escribirlo una vez y controlar mejor las condiciones en las que se ejecuta. Otro consejo que les doy, es que cuando empiecen a escribir un programa, controlen la cantidad de variables que usan, y las variables de las que depende cada parte de su proceso. No queremos llevarnos la sorpresa de que una variable escrita 500 líneas atrás genere un problema en un proceso que no tiene nada que ver. En eso también las funciones nos ayudan muchísimo.

Ya estuvimos usando una función durante toda esta clase: ¡`print()`!. En este caso, las cosas que queremos imprimir por consola se las damos a la función `print` como *argumentos* y `print`, como

si fuera una caja negra de la cuál desconocemos su funcionamiento, imprime esos argumentos en la consola.

Usar funciones ya definidas Cuando usamos una función sin paréntesis tenemos que pensarla como una variable más: una variable del tipo `Callable` (es el tipo de dato especifico de las funciones). Pero cuando le ponemos los paréntesis, como en print, es cuando sucede la magia y la función propiamente "funciona"; o sea que hace las operaciones que tiene definidas. - Hay funciones que no piden ningún argumento, entonces para llamarlas solo hay que ponerle los parentesis vacíos: `funcion_sin_args()` - Hay funciones como `print(arg1, arg2, arg3, ...)` que podemos pasarle uno o muchos argumentos para que imprima por consola - Y hay funciones que además pueden devolver algún valor. Imaginemos una función que suma dos numeros: `sumar(num1, num2)`. La podemos usar de la siguiente forma `resultado = suma(1,2)`. En ese caso `resultado` será igual a 4.

Lectura recomendada: Funciones en Free Code Camp

[97]:
```
# Por ejemplo, esta función base de Python te devuelve el valor más grande de una lista
valor_mas_grande = max([1,2,3,4,5])

print(valor_mas_grande)
```

```
5
```

Definir funciones propias ¡Lo más interesante es que podemos definir funciones que ejecuten el código que nosotros querramos! Para definirlas debemos empezar con la palabra reservada `def`, luego el nombre y los paréntesis. Pero en este caso a lo que está dentro de los paréntesis lo llamamos *parámetros*. Veamos ejemplos:

[98]:
```
# Definamos una función que imprima en consola un mensaje pero que antes nos salude
def saludos_y_mensaje(mensaje): # Esta función tiene un solo parámetro que debera ser in string
    print('Hola '+mensaje)

# Usemos esta función con un mensaje cualquiera
saludos_y_mensaje("Martín") # Le pasamos "Martín" como argumento
```

```
Hola Martín
```

[99]:
```
# Definamos otra función que haga alguna operación aritmética y que nos devuelva el resultado
# Hagamos que haga el promedio de todos los números de una lista

def promedio_de_lista(lista):
    # Tenemos que preguntar por la cantidad de elementos
    cantidad = len(lista) # Python base tiene esta función que devuelve un número entero con el largo de un iterable
```

```
    suma = sum(lista)
    promedio = suma/cantidad
    return promedio # return es la palabra reservada que indica cual es el␣
 ↪valor que la función debe devolver


prom = promedio_de_lista([10,20])
print(prom)
```

```
15.0
```

Como ven, al igual que las estructuras de control de flujo que ya vimos (if, while, for, etc.), el cuerpo de la función también está indentado. En Python la indentación indica un bloque de código, en el caso de las funciones eso implica que los parámetros y cualquier otra variable que definamos adentro de ellas tendrán un alcance local; es decir que solo tienen efecto dentro de la función y no son accesibles por fuera.

[100]:
```
def una_funcion(foo):
    bar = 'leviosa'
    print(foo+' '+bar)

una_funcion('wingardium')
```

```
wingardium leviosa
```

[101]:
```
# Pero la variable bar no existe fuera de la función
print(bar)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/home/luko/indec/curso-python-indec/clase1.ipynb Celda 48 line 2

      <a href='vscode-notebook-cell:/home/luko/indec/curso-python-indec/clase1.
 ↪ipynb#Y100sZmlsZQ%3D%3D?line=0'>1</a> # Pero la variable bar no existe fuera␣
 ↪de la función
----> <a href='vscode-notebook-cell:/home/luko/indec/curso-python-indec/clase1.
 ↪ipynb#Y100sZmlsZQ%3D%3D?line=1'>2</a> print(bar)

NameError: name 'bar' is not defined
```

Esto no es así en el caso contrario: si una variable está definida fuera de una función, esta puede ser usada dentro de ella.

2 FIN

Sé que esta clase fue un montón pero les prometemos que gracias este golpe de contenido las clases que vienen van a ser mucho más divertidas y vamos a poder hacer un montón de cosas.