Python

Índice

Variables

Las variables no exigen un tipo, entonces para declararlas simplemente son `nombre = valor`. Una variable puede cambiar el tipo de dato que contiene

```
x = 1
y = 2
```

```
z = x + y
# z = 3

a = "Hello"
b = "World"
z = a + " " + b
# z = "Hello World"
```

Esto se debe a que Python es un lenguaje de **tipado dinámico**, es decir, los tipos de datos se determinan al correr el programa y no de antes.

Tipos

En Python todo es un objeto con tipo, sin embargo las variables no exigen un tipo.

Se puede checkear el tipo de un valor contenido en una variable usando la función `type()`

```
type(10)
# <class 'int'>
type(x)
# <class 'int'>
type("Python <3")
# <class 'str'>
type(a)
# <class 'str'>
```

Type Casting

Llamando a la clase de un tipo, se puede "castear" un valor de un tipo a otro.

```
int("15")
# 15

int("3f", 16)
# 63

int(15.56)
# 15

float("-11.24e8")
# -11.24e8

bool(x)
# True

str(x)
# "1"

chr(64)
# "A"

bytes([72,9,64])
# b"H\x09@"

list("abc")
# ['a', 'b', 'c']

dict([(3,"three"),(1,"one")])
# {3: 'three', 1: 'one'}

set(["one","two"])
# {'one', 'two'}
```

Condicionales

Los condicionales son unas de las estructuras más básicas de la programación imperativa. Controlan el flujo de ejecución basado en alguna condición. En python, esto se logra mediante if´s

```
if condición:
    # Código si condición es verdadera
else:
    # Código si condición es falsa
```

Donde `condición` es cualquier cosa que devuelva un valor **booleano** (`True` o `False`). Por ejemplo, si quiero ver si un número es más grande que otro,

```
if x > y:
    print(x,"es más grande que",y)
else:
    print(x,"NO es más grande que",y)
```

A veces, uno quiere evaluar más de una condición. Una opción es meter if´s adentro de if´s, pero python nos da otra opción, el `elif` (else if).

```
if condición1:
    # Código si condición1 es verdadera
elif condición2:
    # Código si condición1 es falsa, pero condición2 es verdadera
else:
    # Código si tanto condición1 como condicón 2 es falsa
```

Como ven, el elif tiene sentido cuando la condición depende de alguna forma del primer if. Si no tiene que ver, corresponden if´s separados. Uno puede agregar tantos elif como desee, siguiendo la lógica de que solo se ejecutan si todos las condiciones anteriores son falsas pero la de este elif es verdadera.

Operadores de comparación

Como vimos en el ejemplo anterior, las condiciones deben ser código que devuelva True o False. Una forma muy común de hacer esto son los operadores de comparación, que comparan dos elementos y devuelven verdadero o falso dependiendo dicho operador. Algunos de los más usados:

- `==`: Toma dos valores. Devuelve verdadero si son **iguales**, falso si no. **Es un doble igual. El igual solo es el de asignación de variables**.
- `≠`: Toma dos valores. Devuelve verdadero si son **distintos**, falso si no.
- `>`: Toma dos valores. Devuelve verdadero el valor de la derecha es **mayor** al valor de la izquierda, falso si no.
- `<`: Toma dos valores. Devuelve verdadero el valor de la derecha es **menor** al valor de la izquierda, falso si no.
- `⩾`: Toma dos valores. Devuelve verdadero el valor de la derecha es **mayor o igual** al valor de la izquierda, falso si no.
- `⩽`: Toma dos valores. Devuelve verdadero el valor de la derecha es **menor o igual** al valor de la izquierda, falso si no.

Operadores lógicos

No solo existen operadores de comparación, si no también operadores lógicos, que son operaciones **entre booleanos**.

- `and`: Toma dos expresiones booleanas, devuelve verdadero si **ambos** valores son verdaderos, falso si no.
- `or`: Toma dos expresiones booleanas, devuelve verdadero si **algún** valor es verdadero, falso si ninguno lo es.
- `not`: Toma **una única** expresión booleana, y la invierte. True se convierte en False y viceversa.

Loops

```
x = 0
while x < 10:
    x += 1
    print(x)
```

```
for i in range(10):
    print(i)

for i in [1,2,3,4,5]:
    print(i)

for l in "Hello World":
    print(l)
```

Estructuras Básicas

Listas

Las listas en Python son una estructura de datos que permite guardar muchos elementos en una única variable. Se parecen a los arrays en otros lenguajes, pero con algunas diferencias fundamentales:

- Son de longitud variable, es decir, puedo agregar y sacar elementos de una misma lista
- Los elementos de una lista pueden ser de distinto tipo de datos, por ejemplo, `[4,8,'Mario',5.8]` es una lista válida.

Acceso y asignación

Para acceder a un elemento de una lista, podemos accederlo mediante su posición o índice. **Las listas en Python están indexadas desde 0, no 1.** Por ejemplo, si tengo la siguiente lista:

```
lista = [4,5,0.25,'x']
lista[0] #4
lista[2] #0.25
lista[3] #'x'
```

Puedo no solo acceder, sino también modificar elementos en una lista. Para eso uso operadores de asignación como `=`, o de modificación como `+=` sobre el elemento deseado:

```
lista = [4,5,0.25,'x']
lista[0] = 'Ort'
lista[2] = 1
lista[3] += 'y'
lista == ['Ort',5,1,'xy']
```

Agregar y quitar elementos

Para agregar elementos a una lista en python, podemos usar el método `append`. Un método, por ahora, podemos pensarlo simplemente como una función pero con una sintaxis especial. `append` agrega el elemento pasado por parámetero al final de la lista.

```
lista = [4,5,'x',9]
lista.append(11)
print(lista) #[4,5,'x',9,11]
```

Para agregar un elemento en algun posición específica de la lista se puede usar el método `insert`.

```
lista = [4,5,'x',9]
lista.insert(1, "y")
print(lista) #[4,'y',5,'x',9]
```

Para sacar elementos de la lista, podemos usar el método `pop`, que toma como parámetro el índice que se desea eliminar.

```
lista = [4,5,'x',9]
lista.pop(1)
print(lista) #[4,'x',9]
```

Para sacar un elemento basado en su valor, se puede usar `remove`, que saca la primera aparición del elemento pasado por parámetro.

```
lista = [4,5,'x',9]
lista.remove('x')
print(lista) #[4,5,9]
```

Longitud

Muchas veces cuantos elementos hay en una lista es desconocido, y para eso existe la función `len`.

```
lista = [4,5,'x',9]
print(len(lista)) #4
```

Recorriendo listas

Las listas contienen muchos elementos, y en muchos casos la solución a nuestro problema implica recorrerlas. A continuación discutimos algunas formas de hacer esto.

Mediante índices

Los índices de una lista arrancan en 0 y terminan en la longitud - 1. Por eso, `range(len(l))` nos da los índices de la lista guardada en `l`, ya que range por default arranca en 0 y no incluye el valor final (`len(l)`). Usándolo en un `for`:

```
l = [6,'x',9,25]
for indice in range(len(l)):
    l[indice] #Obtengo los elementos usando el índice para acceder al valor
```

Entiendo que la sintaxis es medio rara y bastante menos intuitiva que un lenguaje como `C#`, pero entiendan que el `for` necesita por parámetro estos iterables, y por eso el `range`.

Mediante elementos

Otras veces no necesitamos la posición del elemento, y podemos aprovechar la sintaxis del `for` a nuestro favor:

```
l = [6,'x',9,25]
for elemento in l:
    elemento #Va a ser cada elemento de la lista de izquierda a derecha (6,'x', etc)
```

Mediante índices y elementos

La forma más elegante, pero a la vez menos intuitiva, de recorrer listas cuando necesito su posición además de su valor, es mediante `enumerate`. Esta función genera un iterable de pares índice, valor dada una lista, y entonces podemos usar la siguiente sintaxis:

```
l = [6,'x',9,25]
for indice,elemento in enumerate(l):
        indice   #posición actual
        elemento #elemento actual
```

Sublistas

Podemos crear sublistas a partir de los elementos de una lista. Este método se llama **slicing** y se basa en la siguiente estructura `lista[comienzo:final:pasos]`. Si no se indica el numero de pasos, por default se recorre a la lista de 1 por 1. Cabe resaltar que el índice de final no incluye esa posición. Los numeros negativos indican los últimos indices de la lista. El -1 es el último, el -2 el anteúltimo y asi sucesivamente.

```
lista[1:3] #[5,0.25]
lista[1:] #[5,0.25,'x']
lista[:2] #[4,5]
lista[1:-1] #[5,0.25]
lista[0::2] #[4,0.25]
```

Diccionarios

```
x = {'a': 1, 'b': 2}
```

Strings

Substrings

Tal como podíamos hacer con las listas, con los strings podemos crear **substrings** a partir de un string con el método slicing. La estructura se mantiene, la cual es la siguiente `string[comienzo:final:pasos]`.

```
string = "neumático"
string[2:5] #'umá'
string[:2] #'ne'
string[4:] #'ático'
string[2:-3] #'umát'
string[2::2] #'uáio'
```

Otras expresiones

List Comprehensions

```
x = [i for i in range(10)]
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[int(x) for x in ('1','29','-3')]
# [1,29,-3]
```

Dictionary Comprehensions

```
y = {i: i for i in range(10)}
```

Ternary Operator

```
x = 1 if y > 2 else 0
```

Aún mas expresiones

```
':'.join(['toto','12','pswd'])
# 'toto:12:pswd'

"words with spaces".split()
# ['words', 'with', 'spaces']

"1,4,8,2".split(",")
# ['1', '4', '8', '2']
```

Funciones

Las funciones en programación son las mismas que en matemática. Simplemente son estructuras que dada una o más entradas, devuelven una salida asociada. Son una especie de caja negra que hace ciertas cosas con la entrada para devolver una salida apropiada.

Las funciones no son estrictamente necesarias en python. Podríamos construir cualquier programa sin usarlas, pero se vuelve extremadamente engorroso hacerlo. Las funciones traen ciertas ventajas a la hora de programar:

- Abstraen comportamiento
- Reducen código repetido
- Mejoran la legibilidad del código
- Hacen al código más mantenible y mejorable

Declaración

Hay 2 partes en el uso de funciones. Por un lado la **declaración**, que es crear la función y "describir" su comportamiento, y la otra es el uso, donde llamamos la función para una entrada concreta.

Para declarar una función, y las partes de dicha declaración, usemos un ejemplo:

```
def duplica(factor):
    producto = factor * 2
    return producto`
```

Analizemos sus partes:

- `def`: Palabra reservada del lenguaje para decir que estamos declarando una función
- `duplica`: Nombre de la función que estamos declarando
- `factor`: Nombre del parámetro (entrada) de la función. Puedo tener más de un parámetro, van separados por comas.
- `return`: Palabra reservada del lenguaje para decir que estamos devolviendo de una función. Si `return` se omite entonces la función devuelve `None`.

Buenas Prácticas

Al declarar una función, es de buenas prácticas:

- Nombre apropiado para la función.
- Nombres apropiados para los parámetros.
- Un único `return`, y explícito, incluso cuando no deseo devolver nada.
- Parametrizar lo más posible el rol que cumple la función para evitar funciones duplicadas.
- Evitar usar cosas declaradas fuera de la función que no fueron pasadas por parámetro.
- Evitar funciones ultra largas. Partir la función en otras funciones si esto tiene sentido para el problema.

Uso

Las funciones están para ser usadas, y luego de declararlas, puedo usarlas pasándole los parámetros correspondientes. La idea es que los valores que le pasé van a reemplazar los parámetros en la declaración.

```
def resta(minuendo,sustraendo):
    diferencia = minuendo - sustraendo
    return diferencia

resultado = resta(5,2) #3
```

La idea es que el primer parámetro (`5`) va a ser el minuendo, y el segundo (`2`) el sustraendo. Luego, lo que esté como `return` es lo que la función va a devolver. En este caso, el resultado se va a guardar en la variable `resultado`.

A una función le podemos pasar de parámetros otras variables, cuentas, otros llamados de función, etc, además de valores concretos. Lo importante es que estén definidos previamente y que se puedan resolver a valores concretos al momento de llamar la función.

Clases

Las clases son una de las formas fundamentales de programar en python, y son la base de **OOP** (**O**bject **O**riented **P**rogrammming). Son una forma de representar entidades del problemas, objetos del mundo real. Se va a entender mejor en la práctica.

Al igual que funciones, no proveen ninguna funcionalidad nueva, pero simplifican sustancialmente programar. Algunas de sus ventajas:

- Agregan semántica al código, dejando más en evidencia la relación entre el programa y el problema a resolver
- **Modularizan** el código, es decir, permite partir el problema en pedazos más chicos.
- Evitan código repetido
- Permiten que los programas sean más mantenibles y escalables

¡Bastante ambicioso! Tengan en cuenta que esta es una introducción básica, y estas ventajas se ven mejor en el uso y con estrategias más avanzadas

Problema de ejemplo

Para explicar clases y objetos, planteemos un problema. Estamos haciendo una red social, y queremos modelar a los usuarios. Obviamente no vamos a modelar el problema entero.

Declaración de clases

¿Como arrancamos? Para declarar una clase, usamos el keyword `class`. De los usuarios, por ahora, solo queremos el nombre de usuario y la contraseña (no se enojen por la falta de seguridad, es un ejemplo). Como se ve en python:

```
class Usuario:
    def __init__(self, nombre_usuario, contraseña):
        self.nombre_usuario = nombre_usuario
        self.contraseña = contraseña
```

¡Un montón de cosas! Vayamos despacio

- `def __init__`: Esto es un método. Un método no es nada más que una función asociada a una clase/objeto. Como ven, la sintaxis es igual que funciones salvo que están dentro del keyword class. **Siempre tienen como primer parámetro al `self`**, (lo vemos en más detalle en otra sección). ¿Y ese nombre extraño? Es un nombre especial, en la sección de **constructores** lo vamos a ver con más detalle.
- `self.nombre_usuario` y `self.contraseña`: Estos son atributos, que no son nada más que variables asociadas a una clase/objeto. Es decir, a donde vaya el objeto, van los atributos.

¿Sigue siendo críptico? Profundizemos

Constructores e instanciación

Los constructores son un método fundamental. Son esenciales en toda declaración de clases (con algunas excepciones, lo vemos más adelante), y son lo que permite **construir instancias de clase**. ¿Qué es esto? Lo que pasa es que todo muy lindo con la idea abstracta de usuario, pero en la vida real, hay usuarios concretos, todos con un nombre y contraseña particular. El constructor es lo que nos permite pasar a estas instancias concretas, que son las que va a usar el programa.

Los constructores se declaran con `__init__`. Siguiendo con el problema:

```
class Usuario:
    def __init__(self, nombre_usuario, contraseña):
        self.nombre_usuario = nombre_usuario
        self.contraseña = contraseña
usuario1 = Usuario("Chona","contraseña_segura") #Llama al constructor de usuario (implícitamente) con esos datos
usuario2 = Usuario("Julian","1234") #Llama al constructor de usuario con otros datos y crea otra instancia
print(usuario1.nombre_usuario) #Chona
print(usuario2.nombre_usuario) #Julian
```

Se habrán dado cuenta que la forma de acceder a los atributos es de la forma `objeto.nombre_atributo`. Un objeto son estas instancias concretas que hablábamos anteriormente, y son la forma en que se usan las clases, las clases por sí solas no hacen nada. Es como declarar una función y no usarla.

Características importantes de los constructores:

- Cómo todo método, el primer parámetro es el `self`.
- Se tienen que declarar todos los atributos que va a tener el objeto con algún valor (pasado o no por parámetro). Hay formas de agregar atributos nuevos fuera del constructor, pero es propenso a errores.
- Se llaman de forma implícita cuando hago `NombreDeClase(parámetros_del_constructor)`. Se puede llamar explícitamente con `NombreDeClase.__init__(parámetros_del_constructor)`

Self

En muchos casos los métodos no solo dependen de parámetros si no también de características del objeto que lo llamó. Esto es el `self`, y es la forma que tiene la clase de referirse a este objeto. En otros lenguajes suele aparecer como `this`. Veamos un ejemplo

```
class Usuario:
    def __init__(self, nombre_usuario, contraseña):
        self.nombre_usuario = nombre_usuario
        self.contraseña = contraseña
    #Defino un método que devuelve True si la contraseña pasada por parámetro es la correcta, False si no.
    def login(self,contraseña_ingresada):
        return self.contraseña == contraseña_ingresada

usuario1 = Usuario("Chona","contraseña_segura")
usuario2 = Usuario("Julian","1234")
print(usuario1.login("1234")) #False. self es usuario1, cuya contraseña es "contraseña_segura"
print(usuario2.login("1234")) #True. self es usuario2, cuya contraseña es "1234"
```

Como se ve, la forma de llamar a un método es de la forma `Objeto.nombreDeMétodo()`. **El `self` no se pasa por parámetro, se llama de forma implícita**, y toma el valor del objeto que llama al método.

El `self` es siempre el primer parámetro en la declaración de un método, y como se llama implícitamente, llamar un método siempre tiene un parámetro obligatorio menos que en la declaración. El llamado de un método también se dice **mensaje**.

Se le puede cambiar el nombre al self simplemente cambiando el nombre del primer parámetro, pero no es recomendado.

Definiciones

Resumamos un poco lo visto hasta ahora acumulando algunas de las definiciones y términos que estuvimos viendo.

Atributos

Variables asociadas a una clase/objeto. Se acceden como `Objeto.nombreDeAtributo`

Métodos

Funciones asociadas a una clase/objeto. Se acceden como `Objeto.nombreDeMétodo()`. **Siempre tienen como primer parámetro al `self`**, que es implícito. Los llamados a métodos se dicen **mensajes**.

Objetos/Instancias

Son las instancias concretas del problema. Se crean como `NombreDeClase(parámetros_del_constructor)`, y **solo pueden ser creados si existe un constructor** (Método de nombre `__init__`).

Métodos especiales

Así como `__init__` es el nombre del método especial reservado para los constructures, existen otros muy útiles.

`__str__`

¿Que pasa si imprimo un objeto?

```
class Usuario:
    def __init__(self, nombre_usuario, contraseña):
        self.nombre_usuario = nombre_usuario
        self.contraseña = contraseña
usuario1 = Usuario("Chona","1234")
print(usuario1) #<__main__.Usuario object at 0×7f8d244b2c10>
```

¿Qué es eso? Por default, python imprime objetos de esa forma bastante críptica. Si queremos que al imprimir `Usuario` nos de algo apropiado, tenemos que definirle la forma de imprimir nuestras clases. Para ello hay que definir el método `__str__`.

```
class Usuario:
    def __init__(self, nombre_usuario, contraseña):
        self.nombre_usuario = nombre_usuario
        self.contraseña = contraseña
    def __str__(self):
        return "Usuario cuyo nombre es " + self.nombre_usuario
usuario1 = Usuario("Chona","1234")
print(usuario1)
#Usuario cuyo nombre es Chona
```

¡Mucho mejor! El método `__str__` también se llama cuando uso `str()`. **Al definir `__str__` debo retornar un string** (texto). Además, `__str__` no toma parámetros (salvo el self).

Typing

Python 3.6+ permite el modulo `typing` para definir tipos de datos. **El tipado no fuerza tipos a variables o parámetros de funciones, el intérprete los ignora.** Son para uso del programador, ya sea como forma de

documentar/aclarar código o para uso de aplicaciones externas como editores de texto,etc.

Los tipos se aclaran de la forma `: type`

```
age : int = 21
name : str = "John"
people : list = ["John", "Jane", "Bob"]


def greeting(name: str) -> str:
    return 'Hello ' + name
```

La notación `->` está para tipar el tipo de retorno de una función.

Puedo detallar aún más los tipos, por ejemplo

```
people : list[str] = ["John", "Jane", "Bob"]
phone_numbers : dict[str,int] = {"John":43338900,"Jane":73335901}
```

Modules

```
import math
math.sqrt(25)
# 5.0

import random
random.randint(1, 10)
# 8

import PIL.Image as Image
im = Image.open("/assets/images/python.jpg")
im.show()
```

Pytest

Para correr los tests usando `pytest` no se corre simplemente con python, si no que hay que correrlo a través de pytest usando el comando

```
python -m pytest nombre_del_archivo_de_tests.py
```

Para construir un test alcanza con declarar una función que comience con el prefijo `test` y poner uno o más `assert` con las condiciones de pasar dicho test.

Supongamos que tengo una función `suma_dos` que toma un entero y le suma 2. Un posible test de esta función en pytest podría ser:

```
def test_suma_dos():
    resultado_obtenido = suma_dos(3)
    resultado_esperado = 5
    assert resultado_esperado == resultado_obtenido
```