Ruta de aprendizaje Python

Realizado por

Dargy Mogollon

Darío Higuera

Leandro Gutierrez

José Fernando Jiménez

Introducción

Vamos a iniciar un nuevo camino con un nuevo lenguaje, recuerda que todo lo que has aprendido hasta este punto será muy útil para enfrentarte a esta nueva aventura.

- ¿Qué es el Zen de Python?

· El Zen de Python, de Tim Peters

· Hermoso es mejor que feo.

· Explícito es mejor que implícito.

· Lo simple es mejor que lo complejo.

· Complejo es mejor que complicado.

· Plano es mejor que anidado.

· Escaso es mejor que denso.

· La legibilidad cuenta.

· Los casos especiales no son lo suficientemente especiales como para romper las reglas.

· Aunque la practicidad vence a la pureza.

· Los errores nunca deben pasar en silencio.

· A menos que se silencie explícitamente.

· Ante la ambigüedad, rechaza la tentación de adivinar.

· Debería haber una, y preferiblemente solo una, forma obvia de hacerlo.

· Aunque esa forma puede no ser obvia al principio a menos que seas holandés.

· Ahora es mejor que nunca.

· Aunque nunca es a menudo mejor que *correcto* ahora.

· Si la implementación es difícil de explicar, es una mala idea.

· Si la implementación es fácil de explicar, puede ser una buena idea.

· Los espacios de nombres son una gran idea, ¡hagamos más de esos! """

- ¿Cómo instalar Python? (Preferiblemente, desde la página oficial, pero si usan Anaconda también está bien)
- ¿Dónde usan Python y Django?
- **Python** es un lenguaje que se utiliza comúnmente para crear prototipos y desarrollar aplicaciones rápidamente. **Django** le da la velocidad y la potencia de **Python** con muchas características integradas adicionales para ayudar a crear aplicaciones web y APIs mucho más rápido.

Tipado de datos

- Definición de variables y su tipado (¿Es necesario tipar en python?)

Python es un lenguaje de *tipado dinámico*. A menudo de seguro habrás escuchado esto; pero, ¿qué significa? «Tipado» ni siquiera es una palabra válida en el español, sino una adaptación del inglés *typing*. Python también es un lenguaje de *tipado fuerte*. Este concepto no es tan frecuente como el primero, pero es asimismo muy relevante. Vamos a explicarlos.

Un lenguaje de programación tiene un sistema de tipos (esta es una mejor forma de ponerlo en español) dinámico cuando el tipo de dato de una variable puede cambiar en tiempo de ejecución. Python efectivamente es, entonces, un lenguaje de tipado dinámico, pues una variable puede comenzar teniendo un tipo de dato y cambiar en cualquier momento a otro tipo de dato. Por ejemplo:

```
1. a = 5
2. print(a)
3. a = "Hola mundo"
4. print(a)
```

Aquí la variable a es creada con el valor 5, que es un número entero (int). Luego en la tercera línea se asigna el nuevo valor "Hola mundo", por lo cual el tipo de dato cambia a una cadena (str). (No es estrictamente correcto hablar de variables en Python). Podemos comprobar el cambio en la consola interactiva:

```
1. >>> a = 5
```

```
>>> type(a)
<class 'int'>
>>> a = "Hola mundo"
>>> type(a)
<class 'str'>
```

Si Python fuera un lenguaje de tipado estático, la segunda asignación de a con una cadena debería arrojar un error, pues su tipo de dato es entero. Por ejemplo, en C:

```
int a = 5;
a = "Hola mundo"; // ¡Error!
```

Pero aquí, además de comprobar que Python es un lenguaje de tipado dinámico, vemos otra característica: la inferencia de tipos. Al decir a = 5 o a = "Hola mundo", Python es capaz de inferir el tipo de dato de una variable a partir del valor que se le está asignando. La inferencia de tipos no es patrimonio exclusivo de lenguajes con tipado dinámico: C++, por ejemplo, es un lenguaje de tipado estático (igual que C) con la capacidad de inferir los tipos.

Por otro lado, un lenguaje es de tipado fuerte cuando, ante una operación entre dos tipos de datos incompatibles, arroja un error (durante la compilación o la ejecución, dependiendo de si se trata de un lenguaje compilado o interpretado) en lugar de convertir implícitamente alguno de los dos tipos. Python es un lenguaje de tipado fuerte. Por ejemplo:

```
a = 5
b = "7"
print(a + b) # ¡Error!
```

Aquí la tercera línea arroja un error, porque un entero (a) no puede sumarse a una cadena (b). Python podría convertir automáticamente la variable b a un entero o a a una cadena para que la operación tenga éxito; pero no lo hace, porque el sistema de tipos es fuerte. Para realizar esta operación, hay que hacer alguna conversión explícita:

```
a = 5
b = "7"
print(a + int(b))
```

Compárese con el mismo código de JavaScript, que es un lenguaje con tipado débil:

```
1. a = 5
2. b = "7"
3. console.log(a + b) // Se ejecuta correctamente.
```

Aquí también a y b tienen tipos de dato diferentes, pero JavaScript convierte a a una cadena para poder realizar la concatenación, porque su sistema de tipos es débil. Esto suele llamarse coerción de tipos. Así, el resultado es "57" (una cadena).

Ahora bien, a pesar de ser un lenguaje de tipado dinámico, Python soporta opcionalmente un sistema de tipos estático. Usando *anotaciones*, podemos indicar el tipo de dato de una variable al crearla:

```
1. a: int = 5
```

Esta es sintaxis válida de Python. El hecho de que el tipado estático en Python sea opcional quiere decir que el intérprete por sí mismo no arrojará un error en tiempo de ejecución si la variable cambia su tipo de dato:

```
1. a: int = 5
2. print(a)
3. a = "Hola mundo" # Esto no arroja ningún error durante la ejecución.
```

La forma de chequear que los tipos de dato de una variable no cambiarán durante la ejecución del programa es usando herramientas como mypy (desarrollada por Guido van Rossum, creador de Python, y colaboradores) o Pyre (desarrollada por Facebook). Así, si ejecutamos mypy sobre el código anterior (llamémosle tipos.py), el resultado es el siguiente:

```
>mypy tipos.py

tipos.py:3: error: Incompatible types in assignment (expression has type "str",
variable has type "int")

Found 1 error in 1 file (checked 1 source file)
```

Vemos que la tercera línea arroja un error, porque se intenta asignar una cadena a una variable cuyo tipo de dato fue anotado como entero. De hecho, ni siquiera es necesario anotar el tipo de dato de a. El siguiente código provoca el mismo error en mypy:

```
1. a = 5
```

2. **print**(a)
3. a = "Hola mundo"

Esto ocurre porque mypy (igual que Python) infiere que el tipo de dato de a es un entero, pues el primer valor que obtiene es 5. Y si comienza siendo un entero, no puede cambiar luego a una cadena. Si realmente queremos permitir que una variable en nuestro código cambie su tipo de dato durante la ejecución, debemos anotarla como Any:

1. **from** typing **import** Any
2.
3. a: Any = 5
4. **print**(a)
5. a = "Hola mundo"

De esta manera el chequeo resulta exitoso:

>mypy tipos.py

Success: no issues found in 1 source file

En conclusión, Python es un lenguaje de tipado dinámico y fuerte, con la posibilidad de adquirir un sistema de tipado estático vía las anotaciones de tipos (como a: int = 5 o b: str = "Hola mundo") y herramientas de chequeo como mypy o Pyre. El gran beneficio de incluir tipado estático en Python radica en la capacidad de las herramientas de chequeo de detectar errores *antes* de la ejecución del programa, con el solo análisis del código.

¿Existe el tipado estático en Python?

El hecho de que el **tipado estático en Python** sea opcional quiere decir que el intérprete por sí mismo no arrojará un error en tiempo de ejecución si la variable cambia su tipo de dato: a: int = 5 print(a) a = "Hola mundo" # Esto no arroja ningún error durante la ejecución

- Conversión de datos a tipos diferentes
- int (x) Convierte x en un entero

- long (x) Convierte x en un entero largo
- float (x) Convierte x en un número de punto flotante
- str (x) Convierte x a una cadena. x puede ser del tipo float. entero o largo
- hex (x) Convierte x entero en una cadena hexadecimal
- chr (x) Convierte x entero a un caracter
- ord (x) Convierte el carácter x en un entero

Funciones, operadores y ciclos

- Operadores lógicos y de comparación

Operador Descripción

> Mayor que. True si el operando de la izquierda es estrictamente mayor que el de la derecha; False en caso contrario.

>= Mayor o igual que. True si el operando de la izquierda es mayor o igual que el de la derecha; False en caso contrario.

< Menor que. True si el operando de la izquierda es estrictamente menor que el de la derecha; False en caso contrario.

<= Menor o igual que. True si el operando de la izquierda es menor o igual que el de la derecha; False en caso contrario.

== Igual. True si el operando de la izquierda es igual que el de la derecha; False en caso contrario.

!= Distinto. True si los operandos son distintos; False en caso contrario.

Funciones en Python

- ¿Qué son los slices en python?

La expresión slicing hace referencia a la operación por medio de la cual se extraen elementos de una secuencia, tal como una lista o una cadena de caracteres. Dependiendo del caso, los elementos podrían ser consecutivos o podrían estar separados dentro de la secuencia original.

- Bucles (While y for)

El bucle For podría traducirse como “para” y el While como “mientras”. En el caso de for no nos permite realizar un ciclo infinito. A diferencia de while que si nos brinda esa posibilidad, a la vez que desde el comienzo no están declaradas al cantidad de iteraciones a realizar.

Agrupación de datos

- Tuplas

En Python, una tupla es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo. Las tuplas se representan escribiendo los elementos entre paréntesis y separados por comas. Una tupla puede no contener ningún elemento, es decir, ser una tupla vacía.

- Listas

Una **lista en Python** es una estructura de datos formada por una secuencia ordenada de objetos. Los elementos de una **lista** pueden accederse mediante su índice, siendo 0 el índice del primer elemento. La función len() devuelve la longitud de la **lista** (su cantidad de elementos).

- Diccionarios

Un Diccionario es una estructura de datos y un tipo de dato en Python con características especiales que nos permite almacenar cualquier tipo de valor como enteros, cadenas, listas e incluso otras funciones. Estos diccionarios nos permiten además identificar cada elemento por una clave (Key).

- Listas y diccionarios anidados

  Un diccionario anidado en Python es un diccionario dentro de otro diccionario, donde los valores del diccionario exterior también son diccionarios. El siguiente código muestra un ejemplo elemental.

Subimos de nivel

Alternativas a los ciclos

- List Comprehensions

Básicamente **son una forma de crear listas de una manera elegante simplificando el código al máximo**. Como lo oyes, Python define una estructura que permite crear listas de un modo un tanto especial.

- Dictionary Comprehensions

Los diccionarios son poderosas estructuras de datos en **Python** que almacenan datos como pares de claves, siendo esta representada en la siguiente forma: Clave-Valor. La comprensión de diccionarios (o **dict comprehension**) puede ser muy útil en crear nuevos diccionarios basados en diccionarios existentes e iterables

Conceptos avanzados de funciones

- Funciones anónimas: lambda

En Python, una función Lambda se refiere a una pequeña función anónima. Las llamamos “funciones anónimas” porque técnicamente carecen de nombre. Al contrario que una función normal, no la definimos con la palabra clave estándar def que utilizamos en Python

- High Order Functions:

Una función se llama **función de orden superior** si contiene otras funciones como parámetro o devuelve una función como salida, es decir, las funciones que operan con otra función se conocen como funciones de orden superior. Vale la pena saber que esta función de orden superior también es aplicable para funciones y métodos que toman funciones como parámetro o devuelven una función como resultado. Python también admite los conceptos de funciones de orden superior.

Propiedades de las funciones de orden superior:

- Una función es una instancia del tipo de objeto.
- Puede almacenar la función en una variable.
- Puede pasar la función como un parámetro a otra función.
- Puede devolver la función desde una función.
- Puedes almacenarlos en estructuras de datos como tablas hash, listas,…

  - Filter

- Map

- Reduce

· Mapaplica una función a todos los elementos en una lista de entrada. Aquí está el plano:

· Plano

· map(function_to_apply, list_of_inputs)
· La mayoría de las veces queremos pasar todos los elementos de la lista a una función uno por uno y luego recopilar la salida. Por ejemplo:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

· Mapnos permite implementar esto de una manera mucho más simple y agradable. Aquí tienes:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

· La mayoría de las veces usamos lambdas mapasí que hice lo mismo. ¡En lugar de una lista de entradas, incluso podemos tener una lista de funciones!

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
```

```
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

4.2. Filtrar

Como sugiere el nombre, filtercrea una lista de elementos para los que una función devuelve verdadero. Aquí hay un ejemplo corto y conciso:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

El filtro se parece a un ciclo for pero es una función integrada y más rápida.

Nota: si el mapa y el filtro no le parecen hermosos, puede leer sobre list/dict/tuple comprensiones.

4.3. Reducir

Reduce es una función realmente útil para realizar algunos cálculos en una lista y devolver el resultado. Aplica un cálculo continuo a pares secuenciales de valores en una lista. Por ejemplo, si desea calcular el producto de una lista de números enteros.

Entonces, la forma normal en que puede realizar esta tarea en python es usar un bucle for básico:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num

# product = 24
```

Ahora probemos con reduce:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

# Output: 24
```

.

Manejo de errores

- Tipos de errores de Python (Esto es muy importante para hacer el debugging y python es superdescriptivo con que error tienes)

  Manejo de excepciones

  try - except - else

  Para evitar la interrución de la ejecución del programa cuando se produce un error, es posible controlar la exepción que se genera con la siguiente instrucción:

  ```
  try:
      bloque código 1
  except excepción:
      bloque código 2
  else:
      bloque código 3
  ```

  Esta instrucción ejecuta el primer bloque de código y si se produce un error que genera una excepción del tipo *excepción* entonces ejecuta el segundo bloque de código, mientras que si no se produce ningún error, se ejecuta el tercer bloque de código.

  `TypeError` : Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

  `ZeroDivisionError` : Ocurre cuando se itenta dividir por cero.

  `OverflowError` : Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.

  `IndexError` : Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.

  `KeyError` : Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.

  `FileNotFoundError` : Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.

ImportError : Ocurre cuando falla la importación de un módulo.

- Assert statements
  - Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.
  - Assertions are simply boolean expressions that check if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.
  - It is also a debugging tool as it halts the program as soon as an error occurs and displays it.

Manejo de archivos

- ¿Cómo se manipulan archivos en python?

Es un consenso no escrito que Python es uno de los mejores lenguajes de programación iniciales para aprender como novato. Es extremadamente versátil, fácil de leer / analizar y bastante agradable a la vista. El lenguaje de programación Python es altamente escalable y es ampliamente considerado como una de las mejores cajas de herramientas para crear herramientas y utilidades que quizás desee utilizar por diversas razones.

Este artículo cubre brevemente cómo Python maneja uno de los componentes más importantes de cualquier sistema operativo: sus archivos y directorios. Afortunadamente, Python tiene funciones integradas para crear y manipular archivos, ya sea archivos planos o archivos de texto. los io module es el módulo

predeterminado para acceder a los archivos, por lo que no necesitaremos importar ninguna biblioteca externa para las operaciones generales de IO.

Las funciones clave utilizadas para el manejo de archivos en Python son: open(), close(), read(), write() y append(). Sigue en:

https://pharos.sh/manejo-de-archivos-en-python/

Programación Orientada a Objetos

- ¿Cómo se maneja este paradigma en python?

  En Python todo es un objeto. Cuando creas una variable y le asignas un valor entero, ese valor es un objeto; una función es un objeto; las listas, tuplas, diccionarios, conjuntos, … son objetos; una cadena de caracteres es un objeto. Y así podría seguir indefinidamente.

  Pero, ¿por qué es tan importante la programación orientada a objetos? Bien, este tipo de programación introduce un nuevo paradigma que nos permite encapsular y aislar datos y operaciones que se pueden realizar sobre dichos datos.

- Clases

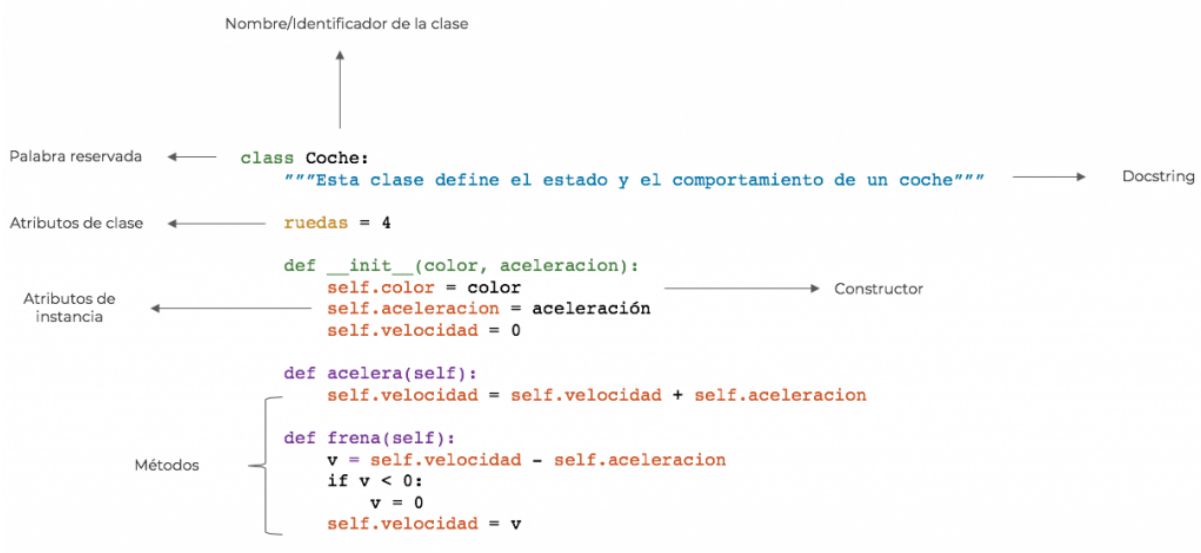
Básicamente, una clase es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Por su parte, un objeto es una concreción o instancia de una clase.

Seguro que si te digo que te imagines un coche, en tu mente comienzas a visualizar la carrocería, el color, las ruedas, el volante, si es diésel o gasolina, el color de la tapicería, si es manual o automático, si acelera o va marcha atrás, etc.

Pues todo lo que acabo de describir viene a ser una clase y cada uno de los de coches que has imaginado, serían objetos de dicha clase.

Para definir una clase en Python se utiliza la palabra reservada `class`. El siguiente esquema visualiza los elementos principales que componen una clase. Todos ellos los iremos viendo con detenimiento en las siguientes secciones:



- Constructor

En la sección anterior me he adelantado un poco… Para crear un objeto de una clase determinada, es decir, *instanciar* una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).

obj = MiClase()

El código anterior crea una nueva instancia de la clase `MiClase` y asigna dicho objeto a la variable `obj`. Esto crea un objeto *vacío*, sin estado.

Sin embargo, hay clases (como nuestra clase `Coche`) que deben o necesitan crear instancias de objetos con un estado inicial.

Esto se consigue implementando el método especial `__init__()`. Este método es conocido como el constructor de la clase y se invoca cada vez que se instancia un nuevo objeto.

El método `__init__()` establece un primer parámetro especial que se suele llamar `self` (veremos qué significa este nombre en la siguiente sección). Pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En nuestro caso, el constructor de la clase coche es el siguiente:

```
def __init__(self, color, aceleracion):
    self.color = color
    self.aceleracion = aceleracion
    self.velocidad = 0
```

Como puedes observar, además del parámetro self, define los parámetros color y aceleracion, que determinan el estado inicial de un objeto de tipo Coche.

En este caso, para instanciar un objeto de tipo coche, debemos pasar como argumentos el color y la aceleración como vimos en el ejemplo:

```
c1 = Coche('rojo', 20)
```

IMPORTANTE: A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un método `__init__()`.

- Getter y Setter (Esto puede parecer denso si no los)

En Python, getters y setters no son los mismos que en otros lenguajes de programación orientados a objetos. Básicamente, el objetivo principal de usar getters y setters en programas orientados a objetos es garantizar la encapsulación de datos. Las variables privadas en python no son en realidad campos ocultos como

en otros lenguajes orientados a objetos. Getters y Setters en python a menudo se usan cuando:

- Usamos Getter y Setter para agregar lógica de validación para obtener y establecer un valor.
- Para evitar el acceso directo a un campo de clase, es decir, las variables privadas no pueden ser accedidas directamente ni modificadas por un usuario externo.

Uso de la función normal para lograr el comportamiento de getters y setters

Para lograr la propiedad getters & setters, si definimos métodos normales get()y set()no reflejará ninguna implementación especial. Por ejemplo

```
# Python program showing a use
# of get() and set() method in
# normal function

class Geek:
    def __init__(self, age = 0):
        self._age = age

    # getter method
    def get_age(self):
        return self._age

    # setter method
    def set_age(self, x):
```

```
        self._age = x


raj = Geek()


# setting the age using setter

raj.set_age(21)


# retrieving age using getter

print(raj.get_age())


print(raj._age)
```

- Decoradores

Hay una forma más de implementar la función de propiedad, es decir, usando decorador . Python @property es uno de los decoradores incorporados. El objetivo principal de cualquier decorador es cambiar los métodos o atributos de su clase de tal manera que el usuario de su clase no necesite realizar ningún cambio en su código. Por ejemplo

```
# Python program showing the use of
# @property

class Geeks:
    def __init__(self):
        self._age = 0

    # using property decorator
    # a getter function
    @property
    def age(self):
        print("getter method called")
        return self._age

    # a setter function
    @age.setter
    def age(self, a):
        if(a < 18):
            raise ValueError("Sorry you age is below eligibility criteria")
        print("setter method called")
        self._age = a

mark = Geeks()
```

```
mark.age = 19


print(mark.age)
```

Extra (Pero importantes)

- Scope (en python)
- Clouseres
- Decoradores
- Si solo investigaste y no hiciste código devuélvete a tratar de programar un **clousere** y un **decorador**
- Sets y operaciones con sets
- Manejo de fechas (***Ya saben con base de datos que si no se tiene claro esto puede ser un dolor de cabeza***)

Bibliografía

https://realpython.com/beginners-guide-python-turtle/

https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf

https://pharos.sh/manejo-de-archivos-en-python/

Python básico:

https://www.youtube.com/watch?v=DLikpfc64cA&t=15592s