Capítulo 3

Python

Al terminar este capítulo, entenderás:

- Cuáles son los tipos de datos básicos de Python.
- Cómo se definen funciones, clases y módulos.
- Cuáles son las estructuras de control.
- Cómo funciona la comprobación opcional de tipos.

Python es un lenguaje interpretado, interactivo y orientado a objetos. Si ya conoces algún lenguaje como Java, la mayoría de lo que encontrarás en Python te sonará familiar. Tiene variables, clases, objetos, funciones y todo lo que se espera que tenga un lenguaje de programación «convencional». Cualquier programador puede empezar a trabajar con Python con poco esfuerzo. Según muchos expertos, Python es uno de los lenguajes más fáciles de aprender y que permiten al novato ser productivo en menos tiempo, y todo eso también explica que sea uno de los lenguajes más populares en la actualidad.

Python es perfecto como lenguaje «pegamento» y para prototipado. Dispone de librerías para desarrollar aplicaciones multihilo, distribuidas, bases de datos, con interfaz gráfica, juegos, gráficos 3D, cálculo científico, análisis de datos, inteligencia artificial y prácticamente todo lo que se te ocurra.

Este pequeño tutorial asume que dispones de Python versión 3.11 o posterior y tienes nociones básicas de programación con algún lenguaje imperativo u orientado a objetos.

3.1. ¡A programar!

Nada como programar para aprender un lenguaje, y en eso Python tiene una ventaja: el intérprete de Python se puede utilizar como una shell (y por eso se dice que es «interactivo»). Es algo tremendamente práctico para probar cosas, pero también para dar tus primeros pasos. Simplemente abre

una consola y ejecuta `python3`, o `python` dependiendo de la configuración de tu SO:

```
$ python3
Python 3.11.9 (main, Apr 10 2024, 13:16:36) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3.2. Variables y tipos

Las variables no se declaran explícitamente, se pueden usar desde el momento en que se inicializan y normalmente no hay que preocuparse por liberar la memoria que ocupan porque viene con un «recolector de basura».

```
$ python3
>>> a = "hola"
```

En el modo interactivo se puede ver el valor de cualquier objeto escribiendo simplemente su nombre:

```
>>> a
'hola'
```

Python es un lenguaje de tipado fuerte. Eso significa que no se puede operar alegremente con variables de tipos diferentes.

```
>>> a + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Esa operación ha provocado que se dispare la excepción `TypeError` que avisa que no hay una forma implícita de convertir el `1` (un entero) para poder utilizar el operador `+` con `a` (una cadena). Las excepciones son el mecanismo más común de notificación de errores.

Sin embargo, puedes cambiar el tipo de la variable `a` sobre la marcha.

```
>>> a = 3
```

En realidad no has cambiado el tipo. Has creado una nueva variable que reemplaza a la anterior.

3.3. Tipos de datos

3.3.1. Valor nulo

Una variable puede existir sin tener un valor, para ello se asigna el valor especial `None`:

3.3.2. Booleanos

Nada fuera de lo común:

```
>>> a = True
>>> a
True
>>> not a
False
>>> a and False
False
>>> 3 > 1
True
>>> b = 3 > 1
>>> b
True
```

3.3.3. Numéricos

Python dispone de tipos enteros de cualquier tamaño que soportan todas las operaciones aritméticas habituales, incluidas las de bits. También hay reales y ambos tipos pueden operar entre sí sin ningún problema. Incluso tiene soporte para números complejos de forma nativa.

```
>>> a = 2
>>> b = 3.0
>>> a + b
5.0
>>> b - 4j
(3-4j)
```

En algunos lenguajes existe el tipo carácter (`char`) que puede manejarse como dato numérico y permite operaciones aritméticas. En Python, sin embargo, se utilizan cadenas de caracteres de tamaño 1 y no permiten operaciones aritméticas.

3.3.4. Secuencias

La secuencia más simple y habitual es la cadena de caracteres (tipo `str`). Las cadenas admiten operaciones como la suma (concatenación) y la multiplicación:

```
>>> cad = 'hola '
>>> cad + 'mundo'
'hola mundo'
>>> cad * 3
'hola hola hola '
```

El tipo **`bytes`** permite manejar secuencias de bytes. Se utilizan habitualmente para serialización de datos. Por ejemplo, puedes codificar una cadena de caracteres a UTF8 como secuencia de bytes y viceversa:

```
>>> word = 'ñandú'
>>> coded = word.encode()
>>> coded
b'\xc3\xb1and\xc3\xba'
>>> coded.decode()
'ñandú'
```

Cuando se utiliza codificación UTF8 algunos caracteres ocupan más de un byte. Este es el caso de las letras 'ñ' y 'ú' del ejemplo. Veremos esta cuestión con más detalle en el capítulo 17.

Las **tuplas** son una agrupación de valores similar al concepto matemático homónimo. Se pueden empaquetar y desempaquetar varias variables, y pueden ser de tipos diferentes.

```
>>> x = cad, 'f', 3.0
>>> x
('hola ', 'f', 3.0)
>>> v1, v2, v3 = x
>>> v2
'f'
```

La tupla, al igual que la cadena, es inmutable, es decir, una vez construida no se puede modificar.

```
>>> cad = 'hola'
>>> cad[0] = 'm'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Si la quieres cambiar, tienes que crear una nueva instancia a partir de la anterior:

```
>>> cad.upper()
'HOLA'
>>> cad
'hola'
```

Las **listas** son similares a los vectores o arrays de otros lenguajes, con la diferencia de que en Python se pueden mezclar tipos. El tipo `list` sí es mutable.

```
>>> x = [1, 'f', 3.0]
>>> x[0]
1
>>> x[0] = None
>>> x
[None, 'f', 3.0]
```

Las listas también se pueden «sumar» y «multiplicar»:

```
>>> x = [1, 'f', 3.0]
>>> x + ['adios']
[1, 'f', 3.0, 'adios']
>>> x * 2
[1, 'f', 3.0, 1, 'f', 3.0]
```

Cualquier tipo de secuencia (listas, tuplas o cadenas) se puede indexar del modo habitual, pero también desde el final con números negativos o mediante «rodajas» (*slicing*):

```
>>> cad = 'holamundo'
>>> cad[-1]
'o'
>>> cad[1:6]
'olamu'
>>> cad[:3]
'hol'
>>> cad[3:]
'amundo'
>>> cad[-6:-2]
'amun'
```

Los **diccionarios** son tablas asociativas (*hash tables*). Tanto las claves como los valores pueden ser de cualquier tipo[1], incluso de tipos diferentes en el mismo diccionario:

```
>>> notas = {'antonio':6, 'maria':9.3, 'juan': 'No presentado'}
>>> notas['antonio']
6
```

Python dispone de otros tipos de datos nativos como conjuntos (`set`), pilas, colas, listas multidimensionales, etc.

3.4. Módulos

Los módulos son análogos a las *librerías* o *paquetes* de otros lenguajes. Para usarlos se utiliza la sentencia `import`:

```
>>> import math
>>> math.cos(math.pi)
-1.0
```

3.5. Estructuras de control

Están disponibles las más comunes: `for`, `while`, `if`, `else`, `break`, `continue`, `return`, etc. Todas funcionan de la forma esperable excepto `for`, que no es un `while` disfrazado. Este `for` itera sobre una secuencia de modo que la

[1] En realidad tiene que ser un tipo *hashable*, es decir, que se pueda asimilar a un valor único

variable de control toma el valor de cada uno de sus elementos en cada iteración; en realidad es más parecido al `for_each` de otros lenguajes:

```
>>> metales = ['oro', 'plata', 'bronce']
>>> for m in metales:
...     print(m)
...
oro
plata
bronce
```

3.6. Indentación estricta

En programación se aconseja «tabular» (indentar) el código para alinearlo con los bloques y así ayudar a su lectura. En Python este estilo es obligatorio porque el esquema de indentación es lo que define los bloques. Es una importante peculiaridad de este lenguaje y obliga a utilizar un editor de código que controle correctamente esta cuestión. Se aconseja tabulación 'blanda' de 4 espacios. Mira el siguiente fragmento de código:

```
total = 0
passed = 0
grades = {'antonio':6, 'maria':9}
for grade in grades.values():
    total += grade
    if grade >= 5:
        passed += 1

print('Average:', total / len(grades))
```

El cuerpo del bloque `for` queda definido por el código que está indentado un nivel debajo de él. Lo mismo ocurre con el `if`. La sentencia que define el bloque siempre lleva dos puntos al final. No se necesitan llaves ni ninguna otra cosa para delimitar los bloques.

3.7. Funciones

Las funciones utilizan una sintaxis muy sencilla e intuitiva. Nada mejor para explicar su sintaxis que un ejemplo:

```
def factorial(n):
    if n == 0:
       return 1

    return n * factorial(n-1)

print(factorial(10))
```

3.8. Python *is different*

Aunque Python se puede utilizar como un lenguaje convencional, siempre hay una manera más «pythónica» de hacer las cosas. Por ejemplo, el soporte de programación funcional que incluye Python permite hacer la función factorial de un modo menos convencional:

```
from functools import reduce
from operator import mul

factorial = lambda x: reduce(mul, range(1, x+1), 1)
print(factorial(10))
```

Si no conoces nada de programación funcional, sirva este ejemplo como mínima introducción. La palabra reservada `lambda` crea una función anónima, que por ejemplo se podría pasar directamente como argumento a otra función. En este caso se almacena en la variable `factorial`, de modo que la podremos usar del mismo modo que la función tradicional del ejemplo anterior. Esa función utiliza la `reduce()`, que es otra herramienta básica de programación funcional. La función `reduce()` aplica una función a los elementos de una secuencia. En este ejemplo, aplica el operador de multiplicación (`mul`) a los enteros del rango `1-x` usando un valor inicial de 1, es decir, multiplica $1 \cdot 1 \cdot 2 \cdot 3 \cdots x$.

3.9. Hacer un 'ejecutable'

Lo habitual en programación es escribir el programa en un archivo de texto y luego compilarlo. Pero como Python es un lenguaje interpretado, no es necesario ningún tipo de procesamiento, al menos no explícito. Simplemente escribe el código en un archivo con extensión `.py`.

Para que la shell sepa qué programa debe ejecutar para interpretar el código de este archivo, es necesario añadir un comentario especial en la primera línea. Esta línea se conoce como «shebang» y tiene el siguiente aspecto:

```
#!/usr/bin/python3
```

Además debes darle permisos de ejecución al archivo:

```
$ chmod +x programa.py
```

Con eso ya puedes ejecutar el programa:

```
$ ./programa.py
```

Al encontrar el shebang, la shell ejecutará el programa como si hubieras escrito `/usr/bin/python3 programa.py`.

En el caso de Python, es más adecuado usar el programa `env` para localizar el intérprete adecuado de Python. El shebang queda así:

```
#!/usr/bin/env python3
```

3.10. Orientado a objetos

Casi todo en Python está orientado a objetos, incluyendo las variables de tipos básicos. ¡Incluso los literales!

```
>>> cad = 'holamundo'
>>> cad.upper()
'HOLAMUNDO'
>>> 'ADIOS'.lower()
'adios'
```

Escribir una clase, al menos algo básico, es muy sencillo:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.bookmarks = []

    def add_bookmark(self, page, label):
        self.bookmarks.append((page, label))

    def last_bookmark(self):
        return self.bookmarks[-1]


book = Book("1984", "George Orwell")
book.add_bookmark(101, "start of chapter 3")
print(book.last_bookmark())
```

Lo más importante es que el primer argumento de los métodos es siempre `self`, que representa la instancia de la clase sobre la que se ejecuta (`Book` en el caso del ejemplo). El método `__init__()` es el constructor[2] y se ejecuta al instanciar la clase: `book = Book("1984", "George Orwell")`.

3.11. *Type checking*

Aunque Python es un lenguaje de tipado dinámico, desde la versión 3.0 se ofrece la opción de hacer «anotaciones de tipo» (*type hints*) sobre variables y argumentos, y además existe un módulo donde se define la semántica y convenciones para especificar de forma opcional esas anotaciones. Un ejemplo:

[2] En realidad es el *inicializador*, pero para un nivel básico nos sirve como idea

```
>>> pi: float = 3.14
```

El comportamiento del lenguaje es el mismo, y las variables pueden «cambiar de tipo» independientemente de su definición sin producir errores:

```
>>> pi: float = 3.14
>>> pi = "Tres coma catorce"
```

¿Y para qué sirve entonces poner el tipo? Principalmente para facilitar la comprensión del código, y sí, también para poder llevar a cabo comprobaciones de tipo (*type checking*), pero para eso se requieren herramientas adicionales como `MyPy`[3].

Veamos el archivo `factorial.py` con una versión de la función de la § 3.7, pero que incluye anotaciones en sus atributos y valor de retorno. También hemos añadido una llamada a la función que toma un argumento de la línea de comandos:

```
import sys

def factorial(n: int) -> int:
  if n == 0:
    return 1

  return n * factorial(n-1)

factorial(sys.argv[1])
```

Si ejecutas `MyPy` para que busque errores en esta implementación, verás lo siguiente:

```
$ mypy factorial.py
factorial.py:13: error: Argument 1 to "factorial" has incompatible type "str"; expected
↪  "int"
Found 1 error in 1 file (checked 1 source file)
```

El programa te avisa que el argumento que estás pasando a la función es una cadena, un tipo de dato que no se corresponde con el esperado (un entero). El error desaparece si transformas el parámetro de entrada del programa en un entero con `int(sys.argv[1])`.

Y ¿qué más?

Esto no ha sido más que una muy breve introducción. Como has podido comprobar Python es un lenguaje muy accesible, con una sintaxis sencilla. Pero eso no significa que sea un lenguaje simple. Aparte de la extensa librería estándar, hay disponible una ingente cantidad de módulos de terceros

[3] `http://mypy-lang.org/`

para casi cualquier necesidad imaginable. Indudablemente es un lenguaje que vale la pena aprender a dominar, con una gran comunidad de usuarios y excelente documentación.