APUNTES RAPIDOS DE PYTHON

Tipos de datos

Los tipos de datos básicos se pueden resumir en esta tabla:

Tipo	Clase	Notas	Ejemplo
str	Cadena en determinado formato...	Inmutable	'Cadena'
bytes	Vector o array de bytes	Inmutable	b'Cadena'
list	Secuencia	Mutable, puede contener objetos...	[4.0, 'Cadena', True]
tuple	Secuencia	Inmutable, puede contener objetos...	(4.0, 'Cadena', True)
set	Conjunto	Mutable, sin orden, no contiene...	{4.0, 'Cadena', True}
frozenset	Conjunto	Inmutable, sin orden, no contiene...	frozenset([4.0, 'Cadena', True])
dict	Diccionario	Grupo de pares clave:valor	{'key1': 1.0, 'key2': False}
int	Número entero	Precisión arbitraria	42
float	Número decimal	Coma flotante de doble precisión	3.1415927
complex	Número complejo	Parte real y parte imaginaria j	(4.5 + 3j)
bool	Booleano	Valor booleano (verdadero o falso)	True o False

- Mutable: si su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.
- Inmutable: si su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Condicionales

Una sentencia condicional ejecuta su bloque de código interno solo si se cumple cierta condición. Se define usando la palabra clave if seguida de la condición y el bloque de código. Si existen condiciones adicionales, se introducen usando la palabra clave elif seguida de la condición y su bloque de código. Las condiciones se evalúan de manera secuencial hasta encontrar la primera que sea verdadera, y su bloque de código asociado es el único que se ejecuta. Opcionalmente, puede haber un bloque final (la palabra clave else, seguida de un bloque de código) que se ejecuta solo cuando todas las condiciones anteriores fueron falsas.

```
>>> verdadero = True
>>> if verdadero:  # No es necesario poner "verdadero == True"
...     print("Verdadero")
... else:
...     print("Falso")
...
Verdadero
>>> lenguaje = "Python"
>>> if lenguaje == "C":  # lenguaje no es "C", por lo que este bloque se obviará y evaluará la siguiente condición
...     print("Lenguaje de programación: C")
... elif lenguaje == "Python":  # Se pueden añadir tantos bloques "elif" como se quiera
...     print("Lenguaje de programación: Python")
... else:  # En caso de que ninguna de las anteriores condiciones fuera cierta, se ejecutaría este bloque
...     print("Lenguaje de programación: indefinido")
...
Lenguaje de programación: Python
>>> if verdadero and lenguaje == "Python":  # Uso de "and" para comprobar que ambas condiciones son verdaderas
...     print("Verdadero y Lenguaje de programación: Python")
...
Verdadero y Lenguaje de programación: Python
COPIA EL CÓDIGO
```

Bucle for

El bucle for es similar a foreach en otros lenguajes. Recorre un objeto iterable, como una lista, una tupla o un generador, y por cada elemento del iterable ejecuta el bloque de código interno. Se define con la palabra clave for seguida de un nombre de variable, seguido de in, seguido del iterable, y finalmente el bloque de código interno. En cada iteración, el elemento siguiente del iterable se asigna al nombre de variable especificado:

```
>>> lista = ["a", "b", "c"]
>>> for i in lista:  # Iteramos sobre una lista, que es iterable
...     print(i)
...
```

```
a
b
c
>>> cadena = "abcdef"
>>> for i in cadena:  # Iteramos sobre una cadena, que también es iterable
...     print(i, end=', ')  # Añadiendo end=', ' al final hacemos que no introduzca un salto de línea, sino una coma y un espacio
...
a, b, c, d, e, f,
```
COPIA EL CÓDIGO

Bucle while

El bucle while evalúa una condición y, si es verdadera, ejecuta el bloque de código interno. Continúa evaluando y ejecutando mientras la condición sea verdadera. Se define con la palabra clave while seguida de la condición, y a continuación el bloque de código interno:

```
>>> numero = 0
>>> while numero < 10:
...     print(numero, end=" ")
...     numero += 1  # Un buen programador modificará las variables de control al finalizar el ciclo while
...
0 1 2 3 4 5 6 7 8 9
```
COPIA EL CÓDIGO

Listas y Tuplas

- Para declarar una lista se usan los corchetes [], en cambio, para declarar una tupla se usan los paréntesis (). En ambas los elementos se separan por comas, y en el caso de las tuplas es necesario que tengan como mínimo una coma.
- Tanto las listas como las tuplas pueden contener elementos de diferentes tipos. No obstante, las listas suelen usarse para elementos del mismo tipo en cantidad variable mientras que las tuplas se reservan para elementos distintos en cantidad fija.
- Para acceder a los elementos de una lista o tupla se utiliza un índice entero (empezando por "0", no por "1"). Se pueden utilizar índices negativos para acceder elementos a partir del final.
- Las listas se caracterizan por ser mutables, es decir, se puede cambiar su contenido en tiempo de ejecución, mientras que las tuplas son inmutables ya que no es posible modificar el contenido una vez creadas.

Listas

```
>>> lista = ["abc", 42, 3.1415]
```

```
>>> lista[0]  # Acceder a un elemento por su índice
'abc'
>>> lista[-1]  # Acceder a un elemento usando un índice negativo
3.1415
>>> lista.append(True)  # Añadir un elemento al final de la lista
>>> lista
['abc', 42, 3.1415, True]
>>> del lista[3]  # Borra un elemento de la lista usando un índice (en
este caso: True)
>>> lista[0] = "xyz"  # Re-asignar el valor del primer elemento de la
lista
>>> lista[0:2]  # Mostrar los elementos de la lista del índice "0" al
"2" (sin incluir este último)
['xyz', 42]
>>> lista_anidada = [lista, [True, 42]] # Es posible anidar listas
>>> lista_anidada
[['xyz', 42, 3.1415], [True, 42]]
>>> lista_anidada[1][0]  # Acceder a un elemento de una lista dentro
de otra lista (del segundo elemento, mostrar el primer elemento)
True
```

COPIA EL CÓDIGO

Tuplas

```
>>> tupla = ("abc", 42, 3.1415)
>>> tupla[0]  # Acceder a un elemento por su índice
'abc'
>>> del tupla[0]  # No es posible borrar (ni añadir) un elemento en
una tupla, lo que provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> tupla[0] = "xyz"  # Tampoco es posible re-asignar el valor de un
elemento en una tupla, lo que también provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupla[0:2]  # Mostrar los elementos de la tupla del índice "0" al
"2" (sin incluir este último)
('abc', 42)
>>> tupla_anidada = (tupla, (True, 3.1415))  # También es posible
anidar tuplas
>>> 1, 2, 3, "abc"  # Esto también es una tupla, aunque es
recomendable ponerla entre paréntesis (recuerde que requiere, al
menos, una coma)
(1, 2, 3, 'abc')
>>> (1)  # Aunque se encuentra entre paréntesis, esto no es una tupla,
ya que no posee al menos una coma, por lo que únicamente aparecerá el
valor
1
>>> (1,)  # En cambio, en este otro caso, sí es una tupla
(1,)
>>> (1, 2)  # Con más de un elemento no es necesaria la coma final
(1, 2)
>>> (1, 2,)  # Aunque agregarla no modifica el resultado
(1, 2)
```

COPIA EL CÓDIGO

Diccionarios

- Para declarar un diccionario se usan las llaves {}. Contienen elementos separados por comas, donde cada elemento está

formado por un par clave:valor (el símbolo : separa la clave de su valor correspondiente).

- Los diccionarios son mutables, es decir, se puede cambiar el contenido de un valor en tiempo de ejecución.
- En cambio, las claves de un diccionario deben ser inmutables. Esto quiere decir, por ejemplo, que no podremos usar ni listas ni diccionarios como claves.
- El valor asociado a una clave puede ser de cualquier tipo de dato, incluso un diccionario.

```
>>> diccionario = {"cadena": "abc", "numero": 42, "lista": [True, 42]}
# Diccionario que tiene diferentes valores por cada clave, incluso una lista
>>> diccionario["cadena"]  # Usando una clave, se accede a su valor
'abc'
>>> diccionario["lista"][0]  # Acceder a un elemento de una lista dentro de un valor (del valor de la clave "lista", mostrar el primer elemento)
True
>>> diccionario["cadena"] = "xyz"  # Re-asignar el valor de una clave
>>> diccionario["cadena"]
'xyz'
>>> diccionario["decimal"] = 3.1415927  # Insertar un nuevo elemento clave:valor
>>> diccionario["decimal"]
3.1415927
>>> diccionario_mixto = {"tupla": (True, 3.1415), "diccionario": diccionario}  # También es posible que un valor sea un diccionario
>>> diccionario_mixto["diccionario"]["lista"][1]  # Acceder a un elemento dentro de una lista, que se encuentra dentro de un diccionario
42
>>> diccionario = {("abc",): 42}  # Sí es posible que una clave sea una tupla, pues es inmutable
>>> diccionario = {["abc"]: 42}  # No es posible que una clave sea una lista, pues es mutable, lo que provocará una excepción
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

COPIA EL CÓDIGO

Sentencia match-case

Python cuenta con la estructura match-case desde la versión 3.10. Esta tiene el nombre de Structural Pattern Matching.

```
match variable:
    case condicion:
        # codigo
    case condicion:
        # codigo
    case condicion:
        # codigo
    case _:
        # codigo
```

COPIA EL CÓDIGO

Cabe destacar que esta funcionalidad es considerablemente más compleja que el conocido switch-case de la mayoría de lenguajes de programación, ya que no solo permite realizar una comparación del valor, si no que también puede comprobar el tipo del objeto, y sus atributos. Además, puede realizar un desempaquetado directo de secuencias de datos, y comprobarlos de forma específica.

En el siguiente ejemplo, se comprueban los atributos de nuestra instancia de Punto. Si en estos no se cumple que x = 10 y y = 40, se pasará a la siguiente condición.

Es importante anotar que Punto(x=10, y=40) no está construyendo un nuevo objeto, aunque pueda parecerlo.

```
from dataclasses import dataclass

@dataclass
class Punto:
    x: int
    y: int

coordenada = Punto(10, 34)

match coordenada:
    case Punto(x=10, y=40):  # los atributos "x" e "y" tienen el valor
especificado
        print("Coordenada 10, 40")
    case Punto():  # si es una instancia de Punto
        print("es un punto")
    case _:  # ninguna condición cumplida (default)
        print("No es un punto")
```

COPIA EL CÓDIGO

En versiones anteriores, existen diferentes formas de realizar esta operación lógica de forma similar:

Usando if, elif, else

Podemos usar la estructura de la siguiente manera:

```
>>> if condicion1:
...     hacer1
>>> elif condicion2:
...     hacer2
>>> elif condicion3:
...     hacer3
>>> else:
```

```
...     hacer
COPIA EL CÓDIGO
```

En esa estructura se ejecutara controlando la condicion1, si no se cumple pasara a la siguiente y así sucesivamente hasta entrar en el else. Un ejemplo práctico sería:

```
>>> def calculo(op, a, b):
...     if op == 'sum':
...          return a + b
...     elif op == 'rest':
...          return a - b
...     elif op == 'mult':
...         return a * b
...     elif op == 'div':
...         return a / b
...     else:
...          return None
>>>
>>> print(calculo('sum',3,4))
7
COPIA EL CÓDIGO
```

Podríamos decir que el lado negativo de la sentencia armada con if, elif y else es que si la lista de posibles operaciones es muy larga, las tiene que recorrer una por una hasta llegar a la correcta.

Usando diccionarios

Podemos usar un diccionario para el mismo ejemplo:

```
>>> def calculo(op, a, b):
...     return {
...         'sum': lambda: a + b,
...         'rest': lambda: a - b,
...         'mult': lambda: a * b,
...         'div': lambda: a/b
...     }.get(op, lambda: None)()
>>>
>>> print(calculo('sum',3,4))
7
COPIA EL CÓDIGO
```

De esta manera, si las opciones fueran muchas, no recorrería todas; solo iría directamente a la operación buscada en la última línea (.get(op, lambda: None)()) y estaríamos dando una opción por defecto. El motivo por el que se usan expresiones lambda dentro del diccionario es para prevenir la ejecución de las instrucciones que contienen a la hora de definir el diccionario. Este únicamente define

funciones como valores del diccionario, y posteriormente, al obtener estas mediante get(), se llama a la función, ejecutando la expresión que esta contiene.

Conjuntos

- Los conjuntos se construyen mediante la expresión set(items), donde items es cualquier objeto iterable, como listas o tuplas. Los conjuntos no mantienen el orden ni contienen elementos duplicados.
- Se suelen utilizar para eliminar duplicados de una secuencia, o para operaciones matemáticas como intersección, unión, diferencia y diferencia simétrica.

```
>>> conjunto_inmutable = frozenset(["a", "b", "a"])  # Se utiliza una lista como objeto iterable
>>> conjunto_inmutable
frozenset(['a', 'b'])
>>> conjunto1 = set(["a", "b", "a"])  # Primer conjunto mutable
>>> conjunto1
set(['a', 'b'])
>>> conjunto2 = set(["a", "b", "c", "d"])  # Segundo conjunto mutable
>>> conjunto2
set(['a', 'c', 'b', 'd'])  # Los conjuntos no mantienen el orden, como los diccionarios
>>> conjunto1 & conjunto2  # Intersección
set(['a', 'b'])
>>> conjunto1 | conjunto2  # Unión
set(['a', 'c', 'b', 'd'])
>>> conjunto1 - conjunto2  # Diferencia (1)
set([])
>>> conjunto2 - conjunto1  # Diferencia (2)
set(['c', 'd'])
>>> conjunto1 ^ conjunto2  # Diferencia simétrica
set(['c', 'd'])
COPIA EL CÓDIGO
```

Listas por comprensión

Una lista por comprensión (en inglés list comprehension) es una expresión compacta para definir listas. Al igual que lambda, aparece en lenguajes funcionales. Ejemplos:

```
>>> range(5)  # La función range devuelve una lista, empezando en 0 y terminando con el número indicado menos uno
[0, 1, 2, 3, 4]
>>> [i * i for i in range(5)]  # Por cada elemento del rango, lo multiplica por sí mismo y lo agrega al resultado
[0, 1, 4, 9, 16]
>>> lista = [(i, i + 2) for i in range(5)]
>>> lista
[(0, 2), (1, 3), (2, 4), (3, 5), (4, 6)]
COPIA EL CÓDIGO
```

Funciones

- Las funciones se definen con la palabra clave def, seguida del nombre de la función y sus parámetros. Otra forma de escribir funciones, aunque menos utilizada, es con la palabra clave lambda (que aparece en lenguajes funcionales como Lisp).
- El valor devuelto en las funciones con def será el dado con la instrucción return.
- Las funciones pueden recibir parámetros especiales para manejar el exceso de argumentos.
    - El parámetro `*args` recibe como una tupla un número variable de argumentos posicionales.
    - El parámetro `**kwargs` recibe como un diccionario un número variable de argumentos por palabras clave.

def:

```
>>> def suma(x, y=2):
...     return x + y  # Retornar la suma del valor de la variable "x" y el valor de "y"
...
>>> suma(4)  # La variable "y" no se modifica, siendo su valor: 2
6
>>> suma(4, 10)  # La variable "y" sí se modifica, siendo su nuevo valor: 10
14
```

COPIA EL CÓDIGO

*args:

```
>>> def suma(*args):
...     resultado = 0
...     # Se itera la tupla de argumentos
...     for num in args:
...         resultado += num #  Suma todos los argumentos
...     return resultado  # Retorna el resultado de la suma
...
>>> suma(2, 4)
6
>>> suma(1, 3, 5, 7, 9)  # No importa el número de variables posicionales que se pasen a la función
25
```

COPIA EL CÓDIGO

**kwargs:

```
def suma(**kwargs):
...     resultado = 0
...     # Se itera el diccionario de argumentos
...     for key, value in kwargs.items():
...         resultado += value  # Suma todos los valores de los argumentos
...     return resultado
...
>>>suma(x=1, y=3)
4
```

```
>>>suma(x=2, y=4, z=6)  # No importa el número de variables por clave que se pasen a la función
12
```

COPIA EL CÓDIGO

```
lambda:
>>> suma = lambda x, y=2: x + y
>>> suma(4)  # La variable "y" no se modifica, siendo su valor: 2
6
>>> suma(4, 10)  # La variable "y" sí se modifica, siendo su nuevo valor: 10
14
```

COPIA EL CÓDIGO

Clases

- Las clases se definen con la palabra clave class, seguida del nombre de la clase y, si hereda de otras clases, los nombres de estas.
- En Python 2.x era recomendable que una clase heredase de object, en Python 3.x ya no hace falta.
- En una clase, un método equivale a una función, y un atributo equivale a una variable.
- **init** es un método especial que se ejecuta al instanciar la clase, se usa generalmente para inicializar atributos y ejecutar métodos necesarios. Al igual que todos los métodos en Python, debe tener al menos un parámetro (generalmente se utiliza self). El resto de parámetros serán los que se indiquen al instanciar la clase.
- Los atributos que se desee que sean accesibles desde fuera de la clase se deben declarar usando self. delante del nombre.
- En Python no existe el concepto de encapsulamiento, por lo que el programador debe ser responsable de asignar los valores a los atributos.

```
>>> class Persona():
...     def __init__(self, nombre, edad):
...         self.nombre = nombre  # Un atributo cualquiera
...         self.edad = edad  # Otro atributo cualquiera
...     def mostrar_edad(self):  # Es necesario que, al menos, tenga un parámetro, generalmente self
...         print(self.edad) # mostrando un atributo
...     def modificar_edad(self, edad):  # Modificando edad
...         if 0 > edad < 150:  # Se comprueba que la edad no sea menor que 0 (algo imposible) ni mayor que 150 (algo realmente difícil)
...             return False
...         else:  # Si está en el rango 0-150, entonces se modifica la variable
...             self.edad = edad  # Se modifica la edad
...
>>> p = Persona('Alicia', 20)  # Instanciando la clase. Como se puede ver, no se especifica el valor de self
```

```
>>> p.nombre  # La variable "nombre" del objeto sí es accesible desde
fuera
'Alicia'
>>> p.nombre = 'Andrea'  # Y por tanto, se puede cambiar su contenido
>>> p.nombre
'Andrea'
>>> p.mostrar_edad()  # Se llama a un método de la clase
20
>>> p.modificar_edad(21)  # Es posible cambiar la edad usando el
método específico que hemos hecho para hacerlo de forma controlada
>>> p.mostrar_edad()
21
"
```