Grado en Ciencia de Datos e Inteligencia Artificial

Asignatura: Algoritmos y Estructuras de Datos

Conceptos Previos

Profesores de Algoritmos y Estructuras de Datos

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Octubre 2020

Contenido

- Introducción
- El tipado en Python
  - Sistemas de tipos
  - Tipado dinámico vs estático
  - Type hints
- El paquete typing
- Ocultación de los detalles de implementación
- Referencias en Python
- Métodos y atributos de clase

Introducción

- En este tema se van a tratar conceptos que son necesarios para entender el resto de la asignatura
  - La mayoría de estos conceptos tienen que ver con la Programación orientada a Objetos.
- Es posible que alguno de ellos ya hayan sido abordados en asignaturas previas
- A lo largo de este tema, se va a introducir también el entorno de desarrollo PyCharm.

Tipado en Python

Sistemas de tipos

- Todo lenguaje de programación implementa algún sistema de tipos
- Esto **permite al lenguaje saber qué puede hacer con un elemento y qué no**
  - Por ejemplo 35 sería un tipo de dato numérico y sobre el se podrían hacer operaciones numéricas
  - Por el contrario “35” sería un tipo de dato cadena de caracteres y las operaciones sobre el serían distintas
- Para manejar estas características los lenguajes de programación se sustentan sobre un sistema de tipado
- Este sistema puede ser más o menos laxo

Tipado dinámico

- El tipado en Python **es dinámico:**
  - El **tipo de una variable se fija con la asignación del dato**
  - El tipo puede variar durante la ejecución de un programa
  - Da mucha flexibilidad a la hora de programar
  - Es una fuente enorme de errores ya que accidentalmente se puede cambiar el tipo de una variable

```
data = 35
print(f" type of data is: {type(data)}")
data = "35"
print(f" type of data is: {type(data)}")
```

```
type of data is: <class 'int'>
type of data is: <class 'str'>
```

Tipado estático

- El tipado en otros lenguajes como Java, C, C++, C# … es un **tipado estático**
- Este tipado se caracteriza por **establecer el tipo de la variable en el momento de la declaración**
- La variable va a mantener ese tipo durante toda la vida de la misma
- Este tipo de tipado previene errores de programación
- A cambio restringe la libertad del programador

```
String name=“Bruce";
String surname=“White";
Date bornDate=Calendar.getInstance().getTime()
```

Type hints en Python

- En un momento determinado de la evolución de Python se considera necesario **dar soporte al control de tipos**
- Pero no se quiere renunciar a la flexibilidad de los tipos dinámicos
- Se introduce lo que se denominó “type hints” en la versión 3.5.0
- Esto no elimina el tipado dinámico de Python
- Permite a ciertos IDEs (p.e. PyCharm) mostrar avisos *(warnings)*, cuando se cambia de tipo una variable
- La herramienta mypy permite detectar cuando se viola el tipo de dato de una variable

Type hints en Python

- Veamos ahora como queda la definición de funciones con *type hints*:

```
def modulo_vector(ux: float, uy: float) -> float:
    return math.sqrt(ux**2 + uy**2)
```

Retorna un float

- Para indicar el valor devuelto por una función o método se usa **->**
- Cuando una función no retorna nada, se debe poner **None**

Type hints en Python

- El siguiente ejemplo ilustra lo que pasaría en PyCharm con los *type hints*:

```
data2: int = 20
print(f" type of data is: {type(data2)}")
data2 = "20"
print(f" type of data is: {type(data2)}")
```

```
type of data is: <class 'int'>
type of data is: <class 'str'>
```

⚠ Expected type 'int', got 'str' instead :9

- Aunque se nos avisa de lo que está pasando, se permite el cambio de tipo.
- Con esta mejora y el IDE adecuado se pueden detectar errores de programación antes de ejecutar el programa.
- En Python el tipo boolean es subtipo de int por lo que no se puede detectar cuando se mezclan estos valores accidentalmente.

El paquete typing

Introducción

- Este paquete está disponible desde la versión 3.5
- Su finalidad es proporcionar un mecanismo para comprobar tipos (“type hints”)
  - Existen herramientas como PyCharm y mypy que permiten estas comprobaciones,
  - pero no detectan todos los posibles errores
- Este paquete introduce el uso de genéricos en Python, la sobrecarga de métodos, etc.
- Define versiones genéricas para las clases contenedoras de Python:

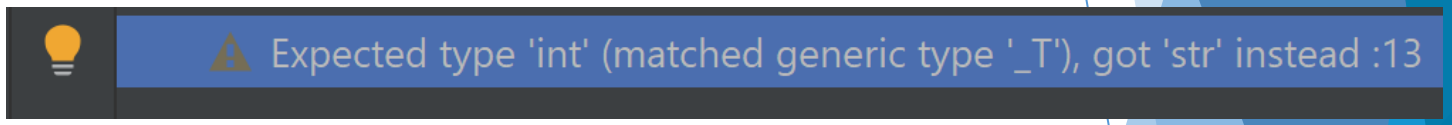
Clase antigua	Clase Genérica
list	List[T]
dict	Dict[K, V]
tuple	Tuple[T1, T2, …]

Clases genéricas en typing

- Las clases contenedoras genéricas permiten indicar el tipo de información que van a contener facilitando el control de errores.

```
from typing import List

data_list: list = []
data_list.append(3)
data_list.append("Hello")
data_list.append(True)
for inf in data_list:
    print(f" data: {inf} type is: {type(inf)}")
data_list2: List[int] = []
data_list2.append(3)
data_list2.append("Hello")
data_list2.append(False)
for inf in data_list2:
    print(f" data: {inf} type is: {type(inf)}")
```




```
mypy typing_example2.py
typing_example2.py:11: error: Argument 1 to "append" of "list"
has incompatible type "str"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```

El paquete typing

- Si se precisa que una variable o un parámetro admita cualquier dato, typing define el tipo **Any**
- Cuando un método o función no retorna nada, typing proporciona la definición de **NoReturn**

```
def func(param: Any) -> NoReturn:
    print(type(param))
```

- Más información en: https://realpython.com/python-type-checking/

Ocultación de los detalles de implementación

Atributos privados

- En los lenguajes orientados a objetos se considera una buena práctica de programación declarar los atributos privados.
  - Si un atributo es privado, solo se puede acceder a él desde los métodos de la propia clase.
  - Esto permite proteger al resto de programa que utilice la clase de futuros cambios en las definiciones de los atributos.
- En Python, un atributo privado se escribe así *__nombre_atributo*
- Para poder acceder a un atributo, se deben definir:
  - **Método getter** (lectura): *atributo() -> tipo*: con el decorador @property,
    - permite *print(objeto.atributo)*
  - **Método setter** (escritura): *atributo(valor: tipo) -> NoReturn*: con el decorador @*nombre_atributo*.setter,
    - permite *objeto.atributo = valor*
- Véase el ejemplo de la clase Point

Métodos privados

- A veces viene bien definir métodos auxiliares en una clase que no tiene sentido que se puedan utilizar fuera de ella.
  - Por ejemplo, para evitar la duplicidad de código
- Estos métodos auxiliares se deben definir como métodos privados
  - Se escriben así *__nombre_método(…) -> tipo:*

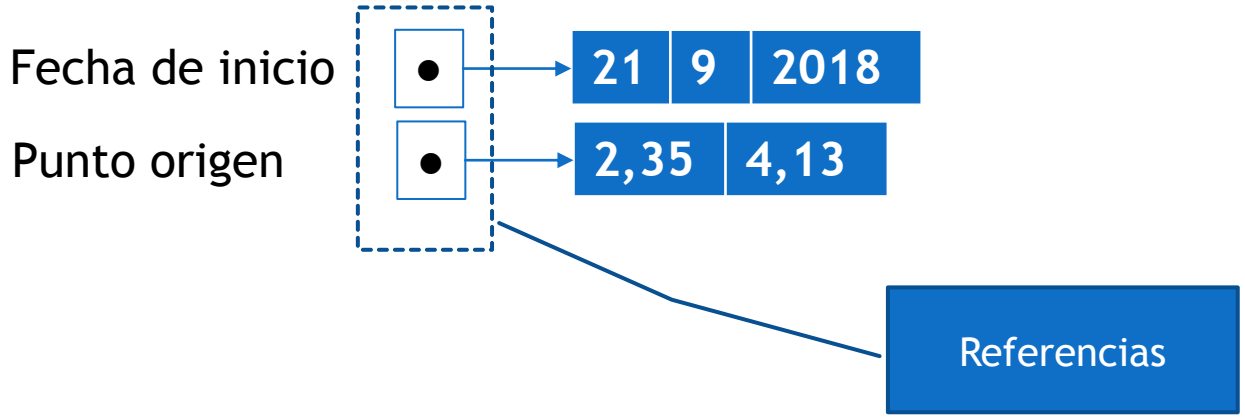
Referencias en Python

Comparación de datos

- Si queremos comparar dos datos de tipos predefinidos (int, bool, str, etc.), utilizaremos el operador de igualdad (**==**)
  - a == 1, b == True, c == “hola”
  - O incluso dos listas l1 == [1, 2]
- Pero qué ocurre si queremos comparar dos objetos
  - En Fundamentos de Programación, habéis utilizado la función esIgual()
  - Pero lo habitual es utilizar la función mágica **__eq__()** para definir el comportamiento del operador == para un nuevo tipo de objetos (sobrecarga del operador)
- Véase el ejemplo de la clase Point con __eq__()

Variables como referencias

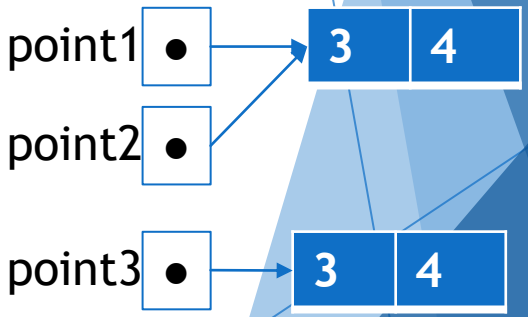
- En Python las variables contienen referencias a los datos de un tipo
- Una referencia se puede ver como un puntero o una dirección de memoria
- Cuando se asigna (=) una variable a otra, se copia la referencia
  - Si queremos obtener una (*shallow copy*) copia de un objeto, utilizaremos el método **copy()** del paquete copy.
- El operador **is** (de identidad) permite comprobar si dos variables contienen la misma referencia



Variables como referencias

- Usamos una clase Point que implementa:
  - __eq__() para la igualdad (==) y __str__() para obtener una representación textual del objeto

```
from point import Point
from copy import copy
point1: Point = Point(3, 4)
point2: Point = point1  # copy reference
point3: Point = copy(point1)
print(f"point1: {point1} point2: {point2}")
# function __eq__ will be used for operator ==. Operator 'is' return true if the references are the same
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
# modify x in point1
point1.x = -3
print("after point1.x = -3")
print(f"point1: {point1} point2: {point2} point3: {point3}")
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
```

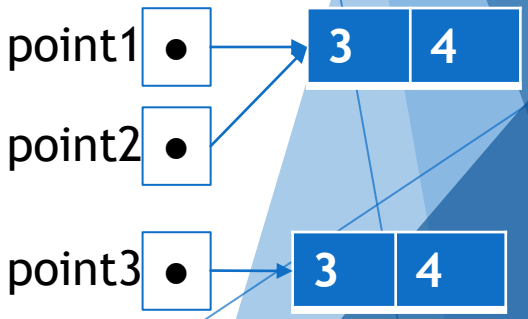


Variables como referencias

- Si ejecutamos el código hasta el punto marcado
  - Por consola tendríamos el resultado de usar los operadores **is** y ==

```
.....
# function __eq__ will be used for operator ==. Operator 'is' return true if the references are the same
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
# modify x in point1
point1.x = -3
print("after point1.x = -3")
print(f"point1: {point1} point2: {point2} point3: {point3}")
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
```

```
point1 is point2: True point3 is point1 False
point1 == point2: True point3 == point1 True
```

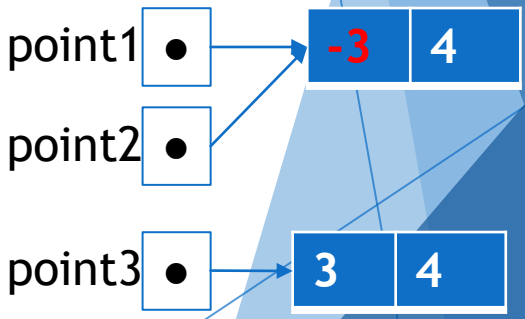


Variables como referencias

- Si ejecutamos el código hasta el punto marcado
  - Se ve como point2 varía por apuntar al mismo objeto que point1
  - Pero point3 permanece inalterado
  - También se ve como el operador is sigue retornando lo mismo

```
.......
# function __eq__ will be used for operator ==. Operator 'is' return true if the references are the same
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
# modify x in point1
point1.x = -3
print("after point1.x = -3")
print(f"point1: {point1} point2: {point2} point3: {point3}")
print(f"point1 is point2: {point1 is point2} point3 is point1 {point3 is point1}")
print(f"point1 == point2: {point1 == point2} point3 == point1 {point3 == point1}")
```

```
point1: (-3, 4) point2: (-3, 4) point3: (3, 4)
point1 is point2: True point3 is point1 False
point1 == point2: True point3 == point1 False
```



Métodos y atributos de clase

Métodos y atributos de clase

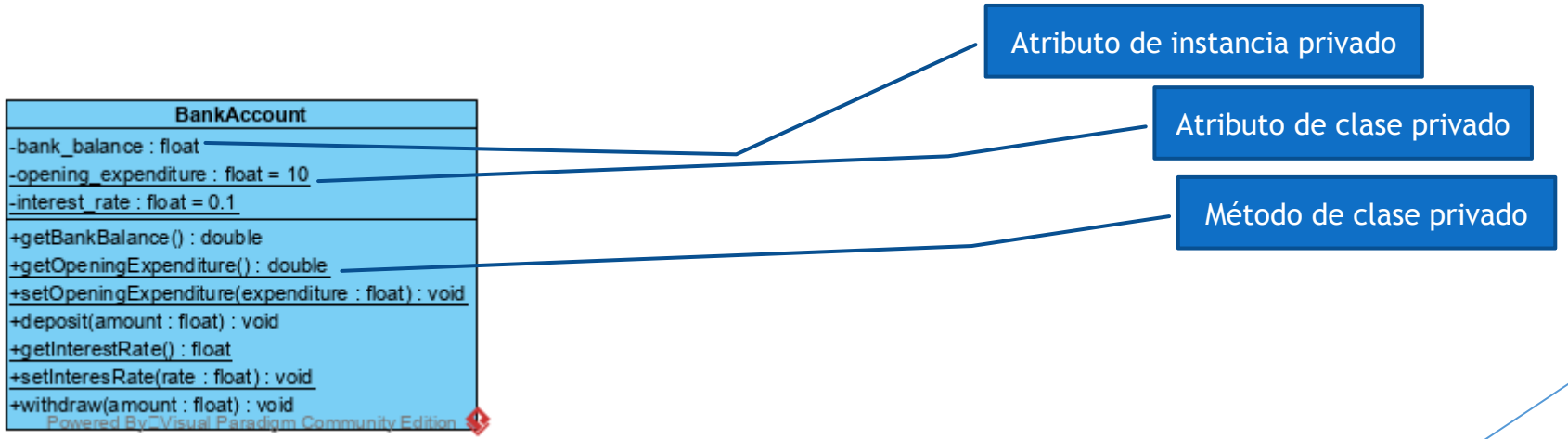
- Hasta ahora se han visto los atributos y métodos de instancia
  - Los valores son específicos de cada objeto/instancia
  - En Python los métodos llevan un parámetro inicial *self*, que hace referencia al objeto sobre el que se está ejecutando el método
    - Por eso los accesos a los atributos de instancia se escriben *self.nombre_atributo*
- Hay programas en el que **todos los objetos de una misma clase deben compartir información**
  - Esa información se guardará en atributos de clase
  - Para **acceder a los atributos de clase**, se utilizarán métodos de clase
    - También se pueden utilizar métodos de clase si el método no utiliza los atributos de instancia de la clase (*self*).
- A continuación vamos a desarrollar un ejemplo para ilustrar el uso de atributos y métodos de clase.

Ejemplo de la cuenta bancaria

- Supongamos que queremos hacer un programa que gestiona las cuentas de una entidad bancaria
- Se quiere implementar la clase Cuenta con las siguientes características:
  - Toda cuenta tiene un **saldo** (atributo de instancia)
  - Común a todas las cuentas se tiene (atributos de clase):
    - **Interés anual** que se devenga sobre el saldo medio de la cuenta.
      - Si se cambia el interés que se devenga, el cambio se verá reflejado en todas las instancias
    - **Gastos de apertura**: Coste que se repercute sobre el saldo inicial al abrir una cuenta.

Ejemplo de la cuenta bancaria

- Se dispondrán de sendos métodos de instancia para consultar el saldo, ingresar y sacar dinero de la cuenta
- Se tendrán métodos de clase para consultar y modificar los gastos de apertura y los intereses



Ejemplo de la cuenta bancaria

```python
from typing import NoReturn


class BankAccount:
    """
    This class represents a bank account
    """
    __opening_expenditure: float = 10
    __interest_rate: float = 0.1
    def __init__(self, amount: float):
        self.__bank_balance: float = amount - BankAccount.__opening_expenditure

    @property
    def bank_balance(self):
        return self.__bank_balance

    def deposit(self, amount: float) -> NoReturn:
        self.__bank_balance += amount

    def withdraw(self, amount: float) -> NoReturn:
        self.__bank_balance -= amount
```

Atributo de clase privado

Atributo de instancia privado

Método de clase privado

```python
@staticmethod
def get_opening_expenditure() -> float:
    return BankAccount.__opening_expenditure

@staticmethod
def set_opening_expenditure(expenditure: float) -> NoReturn:
    BankAccount.__opening_expenditure = expenditure

@staticmethod
def get_interest_rate() -> float:
    return BankAccount.__interest_rate

@staticmethod
def set_interest_rate(rate: float) -> NoReturn:
    BankAccount.__interest_rate = rate
```