Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

8 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

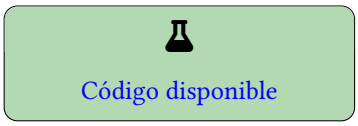
Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Clases

En este capítulo exploraremos algunos conceptos más avanzados sobre Clases, complementando la introducción que hicimos sobre este tema en Introducción a Python ??.



Entraremos en detalle en distintos aspectos de la definición y comportamiento de clases e instancias. Es importante revisar este capítulo incluso si se tiene experiencia con Programación Orientada a Objetos en otros lenguajes, porque algunos términos son muy parecidos (o iguales) pero su comportamiento puede no ser exactamente el mismo.

1.1. De instancia o de clase

En Python podemos definir atributos que serán de instancia o de clase, haciéndose referencia al espacio de nombres en el que está definido dicho atributo.

Para el caso de la instancia, el atributo "vive" dentro de la instancia, y normalmente desde adentro de la instancia los accedemos usando el argumento `self` que se incluye automáticamente en los métodos de instancia:

CELL 01

```
class Power:
    def __init__(self, exp):
        self.exp = exp  # ponemos 'exp' *en la instancia*

    def calc(self, value):
        return value ** self.exp  # usamos 'exp' que vive en la instancia

pow_a = Power(2)
pow_a.calc(5)
```

```
25
```

Desde afuera podemos acceder a ese atributo trabajando con la instancia, y como está definido en el espacio de nombres de cada instancia, si lo modificamos en una, sólo lo modificamos en esa:

CELL 02

```
pow_b = Power(3)
print(f"1. En a: {pow_a.exp}; en b: {pow_b.exp}")
pow_b.exp = 7
print(f"2. En a: {pow_a.exp}; en b: {pow_b.exp}")
```

```
1. En a: 2; en b: 3
2. En a: 2; en b: 7
```

Noten que no podemos acceder a ese atributo desde la clase, porque no tiene ninguna referencia a sus instancias:

CELL 03

```
print("Power tiene 'exp'?", hasattr(Power, "exp"))
```

```
Power tiene 'exp'? False
```

Por otro lado, los atributos de clase "viven" dentro de la clase (están definidos en su espacio de nombre), de la misma manera que los métodos; presten atención a cómo el atributo y el método están definidos en el mismo "lugar":

CELL 04

```
class Power:

    exp = 2

    def calc(self, value):
        return value ** self.exp

p1 = Power()
p1.calc(5)
```

```
25
```

Más allá que esté definido en la clase, cuando se ejecuta el método el valor se puede obtener de la misma forma que antes; esto es porque en el caso de las instancias cuando se buscan en la misma y no se encuentra, la búsqueda continúa en la clase de esa instancia.

Desde afuera podemos acceder al atributo tanto usando la instancia (por el comportamiento antes mencionado) como usando la clase (porque ahora sí está definido allí):

CELL 05

```
print("Desde la instancia:", p1.exp)
print("Desde la clase:", Power.exp)
```

```
Desde la instancia: 2
Desde la clase: 2
```

La característica de que el atributo viva en un sólo lugar (la clase) y no en todas las instancias tiene dos propiedades interesantes: por un lado nos permite ahorrar memoria si la estructura es muy pesada, y por el otro nos permite "compartirla" entre todas las instancias; en otras palabras,

si modificamos el atributo desde la clase vamos a ver desde todas las instancias que se modificó:

CELL 06

```
p2 = Power()
print("Desde la otra instancia:", p2.exp)
```

```
Desde la otra instancia: 2
```

CELL 07

```
Power.exp = 17
print("Desde ambas instancias:", p1.exp, p2.exp)
```

```
Desde ambas instancias: 17 17
```

El efecto de buscar el atributo en la clase cuando no está en la instancia se anula, justamente, cuando se crea el mismo atributo en la instancia. Entonces, vemos que si creamos el `exp` en la primer instancia, esa instancia siempre accederá a ese nuevo valor sin llegar al de la clase:

CELL 08

```
p1.exp = 5
print("Desde esa instancia:", p1.exp)
print("En la clase no cambió:", Power.exp)
print("La otra instancia sigue viendo el de la clase:", p2.exp)
```

```
Desde esa instancia: 5
En la clase no cambió: 17
La otra instancia sigue viendo el de la clase: 17
```

CELL 09

```
Power.exp = 9
print("La primer instancia ve su propio atributo:", p1.exp)
print("La otra instancia sigue viendo el de la clase:", p2.exp)
```

```
La primer instancia ve su propio atributo: 5
La otra instancia sigue viendo el de la clase: 9
```

Pasamos de hablar de atributos a métodos. Aunque la terminología "de clase o de instancia" es similar, el comportamiento es distinto.

Es más, indicar que un método es "de instancia" es un poco confuso: los métodos siempre "viven" en la clase. En este caso, entonces, tenemos que pensar que ese término hace referencia a cómo están vinculada la función definida con la clase, y qué parámetro se incluirá automáticamente al ejecutarla.

Entonces, en el caso de un "método de instancia" cuando es invocado desde la instancia recibirá automáticamente un puntero a la instancia misma, que por convención llamamos *self* ("si mismo" en inglés) en el código:

CELL 10

```
class Prueba:

    def met1(self):
        print("Método de instancia:", self)


pru = Prueba()
pru.met1()
```

```
Método de instancia: <__main__.Prueba object at 0x7f36445230d0>
```

Tengamos en cuenta que si al método "de instancia" lo llamamos desde la clase misma, no habrá ningún parámetro incluido automáticamente, es nuestra responsabilidad pasar explícitamente la instancia que recibirá el método:

CELL 11

```
Prueba.met1(pru)
```

```
Método de instancia: <__main__.Prueba object at 0x7f36445230d0>
```

Alternativamente tenemos los métodos de clase, que recibirán automáticamente un puntero a la clase misma, y métodos estáticos, que no recibirán automáticamente nada extra. En estos casos no importa si son llamados desde la instancia o desde la clase.

Se definen usando los decoradores apropiados:

CELL 12

```
class Prueba:

    def met1(self):
        print("Método de instancia:", self)

    @classmethod
    def met2(cls):
        print("Método de clase:", cls)

    @staticmethod
    def met3():
        print("Método estático!")

pru = Prueba()
pru.met1()
pru.met2()
pru.met3()
Prueba.met1(pru)
Prueba.met2()
Prueba.met3()
```

```
Método de instancia: <__main__.Prueba object at 0x7f3644522560>
Método de clase: <class '__main__.Prueba'>
Método estático!
Método de instancia: <__main__.Prueba object at 0x7f3644522560>
Método de clase: <class '__main__.Prueba'>
Método estático!
```

Los métodos de clase son muy utilizados para implementar métodos alternativos para crear una instancia de la clase. Por ejemplo, en el módulo `datetime` tenemos la clase `date` que al instanciarla normalmente debe recibir el año, mes y día...

CELL 13

```
from datetime import date

date(2022, 12, 18)
```

```
datetime.date(2022, 12, 18)
```

...y alternativamente podemos crear una instancia llamando al método de clase `fromtimestamp` desde la clase misma:

CELL 14

```
date.fromtimestamp(1704715285)
```

```
datetime.date(2024, 1, 8)
```

Copiamos aquí el código real para dicho método (sin el *docstring* y sin el resto de la clase, por cuestiones de espacio) para ayudar a entender cómo trabaja:

```
class date:

```

```
    @classmethod
    def fromtimestamp(cls, t):
        y, m, d, hh, mm, ss, weekday, jday, dst = _time.localtime(t)
        return cls(y, m, d)
```

Vemos que el método recibe la clase como primer parámetro, automáticamente, y luego el *timestamp* que le pasamos. Separa ese valor en año, mes, día, hora, etc, y luego utiliza la misma clase recibida para construir la instancia (usando los valores correspondientes a la fecha), devolviendo justamente eso.

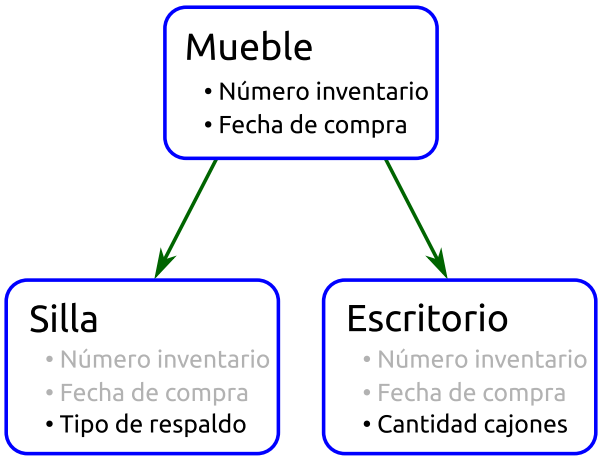
Por otro lado, los métodos estáticos no reciben automáticamente nada (ni cuando se los llama desde la clase ni cuando se los llama desde la instancia). No tienen diferencia con una función "común". Entonces, ¿por qué definirlos como un método estático y no como una función fuera de la clase? En este caso entra en juego una de las características de la programación orientada a objetos que es la responsabilidad: es tarea de esa clase (esos objetos) realizar el cómputo implementado en ese código, entonces es mejor que quede adentro de la clase (como método estático) a que esté "suelto" como función separada.

1.2. Heredando o no

Pasemos a otro concepto que visitamos brevemente en la parte introductoria: la herencia.

Genéricamente hablando, la herencia es un mecanismo de la programación orientada a objetos para reutilizar parte de la implementación del modelado de la realidad. Como las clases hijas adquieren propiedades y comportamientos de las clases ancestras (los "heredan"), también nos referimos a que las clases hijas "especializan" a las ancestras.

Veamos un ejemplo muy sencillo de eso:



En el modelado de la realidad que hicimos para un supuesto sistema de inventario de una oficina, tenemos la clase `Mueble` que tiene atributos genéricos como el número de inventario o la fecha de compra. Elementos más específicos, como el `Escritorio` o la `Silla` tendrán cada uno

sus atributos en particular, pero además heredan los que posee su clase ancestra (ya que los escritorios y las sillas son muebles).

En Python, el concepto clave a entender cuando tenemos una estructura de herencia es que los métodos y atributos no se "pisan" en tiempo de compilación sino que todo se resuelve durante la ejecución del programa.

El siguiente ejemplo nos muestra distintas características que pasaremos a explicar luego, pero básicamente tenemos una clase `Persona` que recibe la fecha de nacimiento (y la guarda como atributo) y tiene un método para calcular la edad, y una clase `Empleado` que hereda de la anterior, que además posee el sueldo y una forma de calcularlo para todo el año.

Cell 15

```
from datetime import date

class Persona:
    def __init__(self, nacimiento):
        self.nacimiento = nacimiento

    def edad(self):
        hoy = date.today()
        return hoy.year - self.nacimiento.year


class Empleado(Persona):
    def __init__(self, nacimiento, sueldo):
        self.sueldo = sueldo
        super().__init__(nacimiento)

    def anual(self):
        return self.sueldo * 12
```

El uso de estas clases es bastante intuitivo:

Cell 16

```
p = Persona(date(1986, 10, 4))
print("Edad p:", p.edad())
e = Empleado(date(1991, 9, 17), 15000)
print("Edad e:", e.edad())
print("Anual e:", e.anual())
```

```
Edad p: 38
Edad e: 33
Anual e: 180000
```

Notemos como usamos el método `edad` en el objeto tipo `Empleado`, ya que esa clase lo "hereda" de `Persona`. Como mencionamos recién, ese método no se "trae" o "copia" de una clase a la otra, sino que es una cuestión de cómo se "resuelve una búsqueda". Cuando hacemos `e.edad` se trata de resolver el atributo `edad` en la instancia y en su clase, y al no encontrarse se empieza a recorrer sus "ancestras" (en este caso de forma trivial ya que hereda de una sola clase).

Por otro lado, el método `__init__` en `Empleado` "pisa" u "oculta" al método con el mismo nombre en `Persona`. Entonces cuando hacemos instanciamos `Empleado` se ejecutará el método de inicialización de Empleado, y no hay ningún mecanismo automático que ejecute otro método

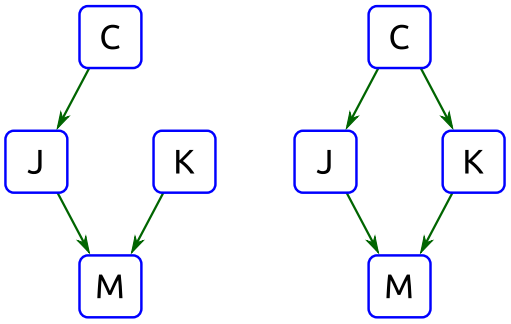
con el mismo nombre en sus ancestras; si queremos que eso suceda debemos hacerlo de forma explícita (como es el caso de nuestro ejemplo, donde inicializamos `Empleado` guardando el sueldo y luego queremos que se termine de inicializar `Persona` con la fecha de nacimiento. Por otro lado, si queremos que el método de la clase hija reemplace o substituya completamente el método de las ancestras, no hace falta realizar ninguna acción, ya que es el comportamiento por defecto.

Como `Empleado` hereda de `Persona`, si queríamos ejecutar el método de inicialización de esta última era suficiente hacer `Persona.__init__(self, nacimiento)` (notar como debemos incluir explícitamente el `self`, puntero a la instancia, porque estamos usando el método directamente desde la clase, como ya vimos en esta sección). Sin embargo es buena costumbre no hacerlo así directamente, ya que esta estructura de dos clases podría crecer y complicarse a futuro y ya no ser tan evidente cual es el próximo ancestro que le corresponde ejecutar el mismo método.

Entonces Python nos provee de una función integrada, `super`, que nos saca el trabajo de determinar cual es la próxima clase ancestra, más allá de como vaya cambiando la estructura (y además nos evitamos tener que pasar el `self` explícitamente). `super` devuelve un objeto intermediario que busca en las clases del camino de herencia, pero en realidad la determinación de cuales son esas clases no depende de `super` mismo sino que está especificado en el atributo `__mro__` de las clases (por *method resolution order*, orden de resolución de métodos).

Debemos entender que no siempre es sencillo darnos cuenta del camino correcto para ir recorriendo las clases ancestras, especialmente cuando dejamos de tener herencia "lineal" y empezamos a encontrar "rombos".

Veamos cómo es el orden en los siguientes dos casos:



Caso 1 Caso 2

El código correspondiente para el primer caso es el siguiente:

CELL 17

```
class C:
    pass

class J(C):
    pass

class K:
    pass

class M(J, K):
    pass

for obj in M.__mro__:
    print(obj)
```

```
<class '__main__.M'>
<class '__main__.J'>
<class '__main__.C'>
<class '__main__.K'>
<class 'object'>
```

Vemos que el contenido del atributo `__mro__` nos indica que luego de `M` tenemos a `J` (primera clase ancestra), luego `C` y `K`, y finalmente `object` (que es el tipo especial de la cual derivan todos los objetos en Python).

Para ir al segundo caso modificamos a la clase `K` (para que también herede de `C`):

CELL 18

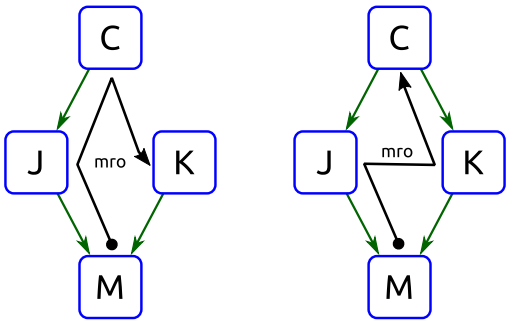
```
class K(C):
    pass


class M(J, K):
    pass


for obj in M.__mro__:
    print(obj)
```

```
<class '__main__.M'>
<class '__main__.J'>
<class '__main__.K'>
<class '__main__.C'>
<class 'object'>
```

Vemos que aunque no modificamos `M`, ¡cambió el orden de resolución de la búsqueda! Los dos caminos son:



Caso 1 Caso 2

Corolario: siempre usemos `super`.

1.3. Usando propiedades

A nivel de la interfaz que se usan en las clases a la hora de construir objetos modelando la realidad, Python difiere un poco de otros lenguajes más rígidos. Por ejemplo, en otros ámbitos es normal tener los atributos protegidos dentro de la instancia (no se los puede acceder ni modificar directamente) e implementer *getters* y *setters* para ello.

Emulando esta misma modalidad tendríamos:

Cell 19

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    def get_nombre(self):
        return self.nombre

    def set_nombre(self, value):
        self.nombre = value

    def get_nacimiento(self):
        return self.nacimiento

    def set_nacimiento(self, value):
        self.nacimiento = value
```

CELL 20

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
p.set_nombre("Carlos")
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15
```

Si en lugar de dos atributos tenemos cinco o diez ya se pueden dar una idea de la cantidad de código "inútil" a la que nos enfrentamos.

En realidad, este código con todos los *getters* y *setters* tiene su motivo de ser. Tenemos que pensar qué pasa si luego de construir la clase y empezarla a usar en nuestros sistemas, de repente entendemos que debemos ejecutar código al leer o escribir algún atributo. Pueden haber mil razones para esto, adaptaciones de valores, validaciones de formatos, cambios de estructuras, etc. Por ejemplo, lo siguiente no queremos que sea posible, ¡debemos validar que la fecha de nacimiento esté en el pasado!

CELL 21

```
p.set_nacimiento(date(2980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
```

```
Persona Carlos del 2980-12-15
```

Como tenemos los *getters* y *setters*, agregar la validación es trivial, y no tenemos que tocar nada de todo el código que usa nuestra clase:

CELL 22

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    def get_nombre(self):
        return self.nombre

    def set_nombre(self, value):
        self.nombre = value

    def get_nacimiento(self):
        return self.nacimiento

    def set_nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self.nacimiento = value
```

CELL 23

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
p.set_nombre("Carlos")
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
p.set_nacimiento(date(2980, 12, 15))
print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15

ValueError                                Traceback (most recent call last)
Input In [23], in <cell line: 5>()
      3 p.set_nombre("Carlos")
      4 print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")
----> 5 p.set_nacimiento(date(2980, 12, 15))
      6 print(f"Persona {p.get_nombre()} del {p.get_nacimiento()}")

Input In [22], in Persona.set_nacimiento(self, value)
     16 def set_nacimiento(self, value):
     17     if value > date.today():
---> 18         raise ValueError("No puede nacer en el futuro")
     19     self.nacimiento = value

ValueError: No puede nacer en el futuro
```

La forma pythonica de armar una clase como esa es exponiendo los atributos internos, que se acceden directamente desde afuera:

CELL 24

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento
```

CELL 25

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nombre = "Carlos"
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nacimiento = date(2980, 12, 15)
print(f"Persona {p.nombre} del {p.nacimiento}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15
Persona Carlos del 2980-12-15
```

Notemos como tanto la definición como el uso de esa clase son más limpios y breves. Pero claro, es evidente ahora la situación de querer agregar la validación que necesitamos (o, en forma genérica, ejecutar código específico en el momento de leer o escribir un atributo). ¿Cómo hacemos eso sin modificar todo el código que usa la clase? Para ello Python tiene las "propiedades".

A través de `property`, integrada en el intérprete, podemos definir propiedades en las clases:

declaraciones que podemos utilizar para ejecutar métodos al momento de (en vez de) leer o escribir un atributo, o incluso borrarlo.

La forma clásica de hacerlo es escribir el *getter* y el *setter* para el atributo, métodos que se ejecutarán cuando lo leamos y escribamos y que internamente van a utilizar otro nombre (normalmente el nombre original con un guión bajo al principio), y luego definir al nombre original del atributo como una propiedad indicando los métodos recién definidos:

CELL 26

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    def _get_nacimiento(self):
        return self._nacimiento

    def _set_nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self._nacimiento = value

    nacimiento = property(_get_nacimiento, _set_nacimiento)
```

Vemos que ahora el "resto del código" queda igual que antes, usando los atributos directamente, y tenemos toda la funcionalidad que deseábamos:

CELL 27

```
p = Persona("carlos", date(1980, 12, 15))
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nombre = "Carlos"
print(f"Persona {p.nombre} del {p.nacimiento}")
p.nacimiento = date(2980, 12, 15)
print(f"Persona {p.nombre} del {p.nacimiento}")
```

```
Persona carlos del 1980-12-15
Persona Carlos del 1980-12-15

ValueError                                Traceback (most recent call last)
Input In [27], in <cell line: 5>()
      3 p.nombre = "Carlos"
      4 print(f"Persona {p.nombre} del {p.nacimiento}")
----> 5 p.nacimiento = date(2980, 12, 15)
      6 print(f"Persona {p.nombre} del {p.nacimiento}")

Input In [26], in Persona._set_nacimiento(self, value)
     10 def _set_nacimiento(self, value):
     11     if value > date.today():
---> 12         raise ValueError("No puede nacer en el futuro")
     13     self._nacimiento = value

ValueError: No puede nacer en el futuro
```

Mediante `property` también podemos definir un método a ejecutar en caso de querer borrar

el atributo, e incluso una cadena de documentación (*docstring*).

Una ventaja de las propiedades es que ese método que ejecuta automáticamente también se dispara cuando se hace la asignación en el método de inicialización. De esta manera tenemos "gratis" también la validación en tiempo de instanciación:

CELL 28

```
Persona("Juan", date(2100, 12, 15))
```

```
ValueError                                Traceback (most recent call last)
Input In [28], in <cell line: 1>()
----> 1 Persona("Juan", date(2100, 12, 15))

Input In [26], in Persona.__init__(self, nombre, nacimiento)
      3 def __init__(self, nombre, nacimiento):
      4     self.nombre = nombre
----> 5     self.nacimiento = nacimiento

Input In [26], in Persona._set_nacimiento(self, value)
     10 def _set_nacimiento(self, value):
     11     if value > date.today():
---> 12         raise ValueError("No puede nacer en el futuro")
     13     self._nacimiento = value

ValueError: No puede nacer en el futuro
```

Entonces no hace falta modificar ninguna parte del sistema que usa nuestra clase, y tenemos la funcionalidad deseada, sin tener que pagar el costo de llenar todo de *getters* y *setters* de forma anticipada por las dudas.

Alternativamente, una forma de utilizar `property` es mediante decoradores; la funcionalidad obtenida es la misma, es sólo una cuestión de gustos. Notar la diferencia entre el decorador para la lectura del atributo, que usa `property`, y el otro para su escritura, que usa el nombre recién definido; ambos métodos usan el nombre del atributo que queremos trabajar (en el primer caso es necesario, en el segundo es por convención):

CELL 29

```
class Persona:

    def __init__(self, nombre, nacimiento):
        self.nombre = nombre
        self.nacimiento = nacimiento

    @property
    def nacimiento(self):
        return self._nacimiento

    @nacimiento.setter
    def nacimiento(self, value):
        if value > date.today():
            raise ValueError("No puede nacer en el futuro")
        self._nacimiento = value
```

Las propiedades también pueden usarse para bloquear el acceso a un atributo. Por ejemplo

en el siguiente caso al no definir un método *setter* hacemos que el atributo no se pueda escribir desde afuera:

CELL 30

```
class Persona:

    def __init__(self, nombre):
        self._nombre = nombre

    def _get_nombre(self):
        return self._nombre

    nombre = property(_get_nombre)
```

CELL 31

```
p = Persona("Juan")
p.nombre
```

```
'Juan'
```

CELL 32

```
p.nombre = "Pedro"
```

```
AttributeError                            Traceback (most recent call last)
Input In [32], in <cell line: 1>()
----> 1 p.nombre = "Pedro"

AttributeError: can't set attribute 'nombre'
```

Es trivial esconderlo totalmente, al no pasarle ningún método a `property`.

En realidad muy pocas veces hacemos esto en Python, ya que tenemos la fuerte convención de usar nombres que comienzan con guión bajo para indicar aquellos métodos o atributos que son privados de un objeto y nunca deberían usarse "desde afuera" (con excepciones, claro, por ejemplo al realizar pruebas de unidad para esos objetos).

1.4. Creando tipos de datos

Cuando necesitamos crear nuevos tipos de datos para utilizar en nuestros programas, lo ideal es que se integren lo mejor posible al lenguaje.

Por ejemplo, supongamos que para nuestro sistema de gestión de centros de datos tenemos una estructura llamada Rack que adentro puede tener uno o más equipos informáticos, entonces podríamos hacer:

CELL 34

```
rack = Rack("power-source", "disks", "cpus", "router")
print("Total de componentes:", rack.length())
print("Posición 1:", rack.get_from_position(1))
print("Alimentación:", rack.power())
```

```
Total de componentes: 4
Posición 1: disks
Alimentación: ['power-source']
```

Aunque esos métodos tienen sentido, son parte de una interfaz que hay que aprender al usar esa clase. En Python hacemos siempre foco en hacer las cosas lo más simple posible y la legibilidad importa, entonces si tenemos la funcionalidad de "obtener el largo" o de "obtener un item de una determinada posición", ¿por qué no proveer los mecanismos para que esas acciones se realicen de la misma manera que es natural en Python?

Entonces, idealmente si nuestro tipo de datos es (por ejemplo) una "colección" de otras cosas, sería natural para los programadores hacer lo siguiente:

CELL 36

```
rack = Rack("power-source", "disks", "cpus", "router")
print("Total de componentes:", len(rack))
print("Posición 1:", rack[1])
print("Alimentación:", rack.power())
```

```
Total de componentes: 4
Posición 1: disks
Alimentación: ['power-source']
```

Por supuesto, no todos los métodos tienen un equivalente en el uso de Python, como es el caso en esta clase cuando devuelve las fuentes de alimentación.

Pero en los casos donde la funcionalidad es equivalente, podemos aprovechar que Python ofrece una especie de protocolo entre la sintaxis usada y la definición de una clase. La idea es más sencilla de lo que parece: Python resuelve las distintas estructuras sintácticas y funcionalidades específicas llamando a determinados métodos en los objetos.

Entonces podemos obtener la misma interfaz que ofrece Python usando esos métodos, llamados métodos con nombres especiales. Son muchos, y en este texto mencionaremos sólo algunos, pero pueden profundizar en su documentación.

Para mostrarlo en el ejemplo, cuando nosotros hacemos `rack[1]` Python ejecuta un método especial (indicándole que buscamos el item en la posición 1), y similar cuando hacemos `len(rack)`.

Veamos la definición de la clase con la integración que buscamos:

CELL 35

```
class Rack:

    def __init__(self, *components):
        self.components = components

    def __len__(self):
        return len(self.components)

    def __getitem__(self, idx):
        return self.components[idx]

    def power(self):
        return [c for c in self.components if c.startswith("power")]
```

Vemos que estos métodos especiales tienen un nombre... especial. Es por eso que hay una cierta ambivalencia con su denominación, a veces se los llama "métodos especiales", otras "métodos de nombre especial", e incluso "métodos mágicos" (pero no hay nada de magia en los mismos, es sólo ese protocolo de Python que antes mencionamos).

Todos estos métodos comienzan con un doble guión bajo y terminan también de esa manera. Por comodidad, se los terminó nombrando con la palabra "*dunder*" (pronunciada "dander"), por *double underscore*. Entonces es normal escuchar "dunder init" por `__init__`, "dunder len" por `__len__`, etc.

Nada evita que los usemos directamente, pero es claramente más elegante lo primero que lo segundo en el siguiente ejemplo:

CELL 37

```
print("Posición 1:", rack[1])
print("Posición 1:", rack.__getitem__(1))
```

```
Posición 1: disks
Posición 1: disks
```

Más allá de la utilización de estos métodos cuando queremos que nuestros objetos se integren a la sintáxis de Python, tambien necesitamos a veces proveer funcionalidad más específica y eso implica la utilización de más de uno de estos elementos.

Para visualizar mejor eso veamos un ejemplo donde tenemos una clase Empleado que queremos guardar luego como clave en un diccionario. Esto, a priori, lo podemos hacer:

CELL 38

```
class Empleado:

    def __init__(self, dni, nombre):
        self.dni = dni
        self.nombre = nombre
```

CELL 39

```
e1 = Empleado("25442321", "Carla")
e2 = Empleado("32250945", "Juan")
aumentos = {e1: 70, e2: 30}
```

Sin embargo es un poco oscuro qué elemento o valor usó el diccionario para poder guardarlo como clave... como el objeto en sí no indicaba nada, usa el identificador del objeto (lo que devuelve la función integrada `id` al aplicarla al objeto). Entonces, si luego construimos otro objeto Empleado en otra parte del código (algo que sucede todo el tiempo, por ejemplo cuando construimos los objetos al momento de necesitarlos con valores que traemos de una base de datos) encontramos que sorpresivamente parecería no estar en la estructura que definimos:

CELL 40

```
e3 = Empleado("25442321", "Carla")
e3 in aumentos
```

```
False
```

Para lograr ese comportamiento nuestro objeto tiene que definir dos métodos especiales: el `__hash__` para cuando se hace `hash` del objeto y el `__eq__` para cuando se lo compara por igualdad (ambas acciones son realizadas por el diccionario para poder guardar el objeto como clave en su estructura interna).

Justamente ese es un punto que tenemos que pensar y analizar de nuestro objeto, ¿qué atributo queremos que sea una clave unívoca? Para el caso del ejemplo es sencillo: usaremos el DNI del empleado. Es trivial agregar entonces los dos métodos en cuestión:

CELL 41

```
class Empleado:

    def __init__(self, dni, nombre):
        self.dni = dni
        self.nombre = nombre

    def __hash__(self):
        return hash(self.dni)

    def __eq__(self, other):
        return self.dni == other.dni
```

CELL 42

```
e1 = Empleado("25442321", "Carla")
e2 = Empleado("32250945", "Juan")
aumentos = {e1: 70, e2: 30}

Empleado("25442321", "Carla") in aumentos
```

```
True
```

Por otro lado, si lo que queremos es que nuestro objeto se comporte completamente como uno de los tipos integrados de Python, tenemos que aprovechar el módulo `collections.abc` donde te-

nemos clases abstractas que funcionan como base de nuestros objetos y nos permiten, definiendo sólo algunos métodos, cumplir con todo el "protocolo estándar" de cada tipo.

Mostremos como ejemplo lo que podríamos denominar un "diccionario con claves en minúscula". Para ello heredamos de `Mapping` (la clase abstracta para "mapas" de sólo lectura) y definimos los tres métodos mínimos necesarios:

CELL 43

```
from collections.abc import Mapping

class Test(Mapping):

    def __init__(self, **data):
        self.data = {key.lower(): value for key, value in data.items()}

    def __getitem__(self, key):
        return self.data[key.lower()]

    def __len__(self):
        return len(self.data)

    def __iter__(self):
        return iter(self.data)
```

Debemos prestar atención en esa definición como siempre llevamos a mínúscula cada clave antes de trabajar el diccionario interno que tenemos. Entonces, luego accedemos a nuestra estructura sin importar la forma de la clave:

CELL 44

```
test = Test(Foo=3, bar=1)
print("Mayúscula:", test["FOO"])
print("Un mix:   ", test["fOo"])
print("Minúscula:", test["foo"])
```

```
Mayúscula: 3
Un mix:    3
Minúscula: 3
```

Lo importante de haber heredado `Mapping` es que nuestro objeto se comporta "en general" como un diccionario de Python aunque no hayamos definido el resto de los métodos:

CELL 45

```
print("Largo:", len(test))
print("Claves:", list(test.keys()))
```

```
Largo: 2
Claves: ['foo', 'bar']
```

Como mencionamos arriba, esto sería para un diccionario de sólo lectura. Para el comportamiento completo debemos trabajar con `MutableMapping` y escribir nosotros un método más (el `__setitem__`, como indica esta tabla resumen.

Recomendamos analicen la documentación de este módulo ya que es una buena forma de

aprender también las distintas interfaces de los tipos de datos incluidos en Python.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [2].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.

VERSIÓN PRELIMINAR