# Metodos Magicos

October 31, 2020

## 1 Métodos mágicos

Los métodods mágicos en Python son métodos especiales que agregan "magia" a las clases. Los métodos mágicos no **son para llamarse directamente**, sino más bien que se invocan indirectamentes. Por ejemplo, existe un método mágico llamado `__add__()` que se invocan cuando se utiliza el operador `+`.

[1]:
```
var = 2
var + 2
```

[1]:
```
4
```

[53]:
```
# var = var + 2
```

[3]:
```
var.__add__(2)
```

[3]:
```
4
```

[4]:
```
var
```

[4]:
```
2
```

Para ver los métodos mágicos de una clse usamos `dir()`

[5]:
```
dir(int)
```

[5]:
```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
```

```
'__floor__',
'__floordiv__',
'__format__',
'__ge__',
'__getattribute__',
'__getnewargs__',
'__gt__',
'__hash__',
'__index__',
'__init__',
'__init_subclass__',
'__int__',
'__invert__',
'__le__',
'__lshift__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmod__',
'__rmul__',
'__ror__',
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
```

```
 '__truediv__',
 '__trunc__',
 '__xor__',
 'as_integer_ratio',
 'bit_length',
 'conjugate',
 'denominator',
 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
```

[50]:
```
dir(list)
```

[50]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
```

```
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

[8]:
```
class Foo:
    pass
```

[9]:
```
dir(Foo)
```

[9]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__']
```

[54]:
```
hash(Foo)
```

[54]: 1940260

[61]:
```
class Foo:
    def __hash__(self):
        print("hola")
        return 1
```

[62]:
```
a = Foo()
```

[63]:
```
hash(a)
```

```
hola
```

[63]: 1

### 1.1 __new__()

El método `__new__()` es llamado después de `__init__()` y retonar un nuevo objeto, que es inicializado en `__init__()`. La instancia puede ser creado dentro del método `__new__` usando la función super.

[68]:
```
class Persona:
    def __new__(cls, *args, **kwargs):
        print("__new__ metodo")
        id_persona = id_persona + 1
        instance = super(Persona, cls).__new__(cls)
        return instance

    def __init__(self, name, age):
        print('__init__ metodo')
        self.name = name
        self.age = age
```

```
hola juan
```

[66]:
```
p = Persona("Juan", 20)
```

```
__new__ metodo
__init__ metodo
```

[69]:
```
print(p.name, p.age)
```

```
Juan 20
```

### 1.2 __str__()

Retorna el string printeable de un objeto definido por el usuario. Este método se ejecuta cuando se usa la función `str()`.

[24]:
```
str(p)
```

[24]:
```
'<__main__.Persona object at 0x7fc2fc43d820>'
```

[25]:
```
print(p)
```

```
<__main__.Persona object at 0x7fc2fc43d820>
```

[74]:
```
class Persona:
    def __new__(cls, *args, **kwargs):
        print("__new__ metodo")
        instance = super(Persona, cls).__new__(cls)
        return instance

    def __init__(self, name, age):
        print('__init__ metodo')
        self.name = name
        self.age = age

    def __str__(self):
        return f'Esta es una clase PErsona id: {id(Persona)}'
```

[75]:
```
p = Persona("Emma", 25)
```

```
__new__ metodo
__init__ metodo
```

[72]:
```
str(p)
```

[72]:
```
'Nombre: Emma. Edad: 25'
```

[78]:
```
print(p)
```

```
Esta es una clase PErsona id: 31312144
```

[79]:
```
repr(p)
```

[79]:
```
'<__main__.Persona object at 0x7fc2fc3860a0>'
```

### 1.3 __repr__()

Representación del tipo del clase

[83]:
```python
class Persona:
    def __new__(cls, *args, **kwargs):
        print("__new__ metodo")
        instance = super(Persona, cls).__new__(cls)
        return instance

    def __init__(self, name, age):
        print('__init__ metodo')
        self.name = name
        self.age = age

    def __str__(self):
        return f'Nombre: {self.name}. Edad: {self.age}'

    def __repr__(self):
        return f'Este es un objeto de Persona'
```

[84]:
```python
p = Persona("Emma", 25)
```

```
__new__ metodo
__init__ metodo
```

[85]:
```python
repr(p)
```

[85]:
```
'Este es un objeto de Persona'
```

[86]:
```python
print(p)
```

```
Nombre: Emma. Edad: 25
```

### 1.4 __getattr__() __setattr__() __delattr__()

`__getattr__()` Se llama cuando trata de acceder a un atributo que no existe. Un buen lugar para el control de errores. `__setattr__()` Se invoca cuando se asigna un valor. `__delattr__()` Se invocoa cuando se elimina.

[128]:
```python
class Foo:
    def __getattr__(self, name):
        # control
        print("te equivocaste ...")
        # ejecutar busqueda de recomendacion
        recomendacion = 'message'
        print(f"Talvez quisiste decir {recomendacion}")
        return recomendacion

    def __delattr__(self, name):
        print("no no elimino lo que me estas pidiendo")
```

[ ]:

[118]:
```
foo = Foo()
```

[108]:
```
foo.name = "hola"
```

[97]:
```
print(foo.name)
```

```
hola
```

[109]:
```
print(foo.mensaje)
```

```
te equivocaste …
Talvez quisiste deci message
message
```

[125]:
```
foo.name = None
```

[120]:
```
print(foo.name)
```

```
Emmanuel
```

[126]:
```
del foo.name
```

```
no no elimino lo que me estas pidiendo
```

[127]:
```
print(foo.name)
```

```
None
```

### 1.5 Operadores mágicos

[226]:
```
# LIFO

# pila + elemento <- pila.append(elemento)
# pila > elemento <- elemento = pila.pop()

class Element:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return str(self.value)

class Pila:
    def __init__(self):
        self.pila = list()
```

```
    def __add__(self, element):
        # Append(elemento)
        self.pila.append(element)

    def __gt__(self, element):
        # greater than
        try:
            element = self.pila.pop()
        except:
            print("No more element")
        return True

    def mostrar(self):
        print(self.pila)
```

[228]:
```
pila = Pila()
```

[229]:
```
pila.mostrar()
```

```
[]
```

[230]:
```
element = Element(10)
```

[235]:
```
pila + element
```

[236]:
```
pila.mostrar()
```

```
[10, 10, 10, 10, 10]
```

[247]:
```
pila > element
```

```
No more element
```

[247]:
```
True
```

[248]:
```
pila.mostrar()
```

```
[]
```

[249]:
```
element.value
```

[249]:
```
10
```

## 2 Ejercicio

Crear una clase Processor, que ejecuta un script (implementar si se quiere) y que tenga cómo mínimo, una variable `status` y `name`, que guarde el estado de un proceso: “waiting”, “running” o

"finished". A cada proceso yo puedo definir los datos que quiera, es decir que no estén predefinidos.

Los precesos son gestionados por un Manager, dónde yo puedo agregar cada a proceso a la lista de procesos con un `+` y eliminarlos con un `-`. Cambiar el estado con usando `>` por ejemplo `Proceso1 > waiting`.

Solo agregar a la lista Procesos que no existan. Y si se trata de ejecutar un proceso que no existe lanzar una exception.

Si yo hago `len(Manager)` quiero saber la cantidad de procesos.

+ lo que quieran

[165]:
```
class Processor:
    def __init__(self):
        self.status = ''

class Manager:
    def __init__(self):
        self.m = []


mananger = Manager()

manager.m = [procesosr.name= "hola", processor.name = "chau", processor.name =␣
↪"huancion"]
```

```
          File "<ipython-input-165-498a02df4f11>", line 12
        manager.m = [procesosr.name= "hola", processor.name = "chau", processor.
↪name = "huancion"]
                                             ^
    SyntaxError: invalid syntax
```

[164]:
```
processor.name = "huancion"

manager.m.procecsor.run()

procesosr > "running"
```

```
hola
```