Python Avanzado

El nombre mas pretencioso que escucharan hoy

Juan-Pablo Silva
jpsilva@dcc.uchile.cl

Contenidos

Conceptos Generales

*args y **kwargs

- Unpacking
- Permiten un numero arbitrario de argumentos
- Filtrar argumentos con **kwargs

Unpacking

In [ ]:
```
lista = [1, 2, 3, 4]

a1, a2, a3, a4 = lista
print(a2)
```

In [ ]:
```
print(lista)
print(*lista)
```

Mas general

In [ ]:
```
def f1(*args):
    print(args)
    print(*args)

def f2(**kwargs):
    print(kwargs)
    print(*kwargs)
    print(", ".join(["{}={}".format(k,v) for k,v in kwargs.items()]))

def f3(*args, **kwargs):
    print(args, kwargs)
```

In [ ]:
```
print("-----F1------")
f1(1,2,3,4,5)
print("-----F2------")
f2(a=10, b=20, c=30)
print("-----F3------")
f3(1,2,4, a=20, r=10, s="un String")
```

Unir diccionarios

- usamos **kwargs

In [ ]:
```
%reset -f
```

In [ ]:
```
a = {
    "a":10,
    "b":20
}

b = {
    "b":100,
    "c":200
}

print(a)
print(b)
```

In [ ]:
```
c = {**a, **b}
print(c)
d = {**b, **a}
print(d)
```

Forzar *keyword*

- funciones/metodos con argumentos opcionales
- prevenir equivocaciones

In [ ]:
```
%reset -f
```

Aceptamos de *todo*

In [ ]:
```
def f1(a1, a2, *args, k1=10, k2=20, **kwargs):
    print(f"a1={a1}, a2={a2}")
    print(f"args: {args}")
    print(f"k1={k1}")
    print(f"k2={k2}")
    print(f"kwargs: {kwargs}")
```

In [ ]:
```
f1(1, 2, 3, k1=20, k2=30, a=100, b=200)
```

Ahora sin *args

In [ ]:
```
def f1(a1, a2, k1=10, k2=20, **kwargs):
    print(f"a1={a1}, a2={a2}")
    print(f"k1={k1}")
    print(f"k2={k2}")
    print(f"kwargs: {kwargs}")
```

In [ ]:
```
f1(1, 2, 3, k1=20, k2=30, a=100, b=200)
```

Por que pasa esto

In [ ]:
```
def f1(a1, a2, k1=10, k2=20):
    print(f"a1={a1}, a2={a2}")
    print(f"k1={k1}")
    print(f"k2={k2}")
```

In [ ]:
```
f1(1, 2, 3, 4)
```

La solucion: *

In [ ]:
```
def f1(a1, a2, *, k1=10, k2=20):
    print(f"a1={a1}, a2={a2}")
    print(f"k1={k1}")
    print(f"k2={k2}")
```

In [ ]:
```
try:
    f1(1,2,3,4)
except TypeError as e:
    print(e)
```

In [ ]:
```
f1(1, 2, k1=3, k2=4)
```

Switch-case

- No existen
- Hay que arreglarselas

In [ ]:
```
%reset -f
```

Usando un diccionario es lo mas simple

In [ ]:
```
def switcher(value):
    options = {
        "val1": 1,
        "val2": 2,
        "val3": 3,
        "val4": 4,
    }
    if value not in options:
        raise ValueError()
    return options[value]
```

In [ ]:
```
print(switcher("val3"))
print(switcher("val1"))
```

Podemos guardar funciones adentro tambien

In [ ]:
```
def f1(): return 1
def f2(): return 2
def f3(): return 3
def f4(): return 4

def switcher(value):
    options = {
        "val1": f1,
        "val2": f2,
        "val3": f3,
        "val4": f4,
    }
    if value not in options:
        raise ValueError()
    return options[value]
```

In [ ]:
```
print(switcher("val3")())   # <-- notar el ()
print(switcher("val1")())   # <-- notar el ()
```

`global`

- cuando algo que funcionaba ya no funciona

In [ ]:
```
%reset -f
```

Esto funciona lo mas bien

In [ ]:

```
def f(a):
    print(a)
    print(b)

f(10)
```

Era broma, ahora si

In [ ]:
```
b = 20

def f(a):
    print(a)
    print(b)

f(10)
```

Peeero ahora ya no funciona

In [ ]:
```
b = 20

def f(a):
    print(a)
    print(b)
    b = 100

f(10)
```

??????????????????????

Sucede que tenemos que definir que `b` es una variable global si vamos a asignarla

In [ ]:
```
b = 20

def f(a):
    global b
    print(a)
    print(b)
    b = 100

f(10)
```

Pero por que?

In [ ]:
```
%reset -f

b = 20
```

In [ ]:
```
def f(a):
    print(locals())

f(10)
```

In [ ]:
```
def f(a):
    import __main__
    print(locals())
    print("b in __main__?", "b" in vars(__main__))
    __main__.b = 100

f(10)
```

`nonlocal`

- cuando deberias estar usando una clase

In [ ]:
```
%reset -f
```

Calculemos el "*running average*"

In [ ]:
```
def average():
    lista = []
    def running_avg(value):
        lista.append(value)
        s = sum(lista)
        return float(s)/len(lista)

    return running_avg

avg = average()
```

In [ ]:
```
print(avg(1))
print(avg(5))
print(avg(10))
print(avg(3))
print(avg(50))
```

Como funciona esto?

Como funciona esto?

- Free variables

In [ ]:
```
print(avg.__code__.co_freevars)
```

In [ ]:
```
print(avg.__closure__)
print(avg.__closure__[0].cell_contents)
```

Igual es ineficiente guardar todo el rato la lista...

In [ ]:
```
def average():
    suma = 0.
    lista = 0
    def running_avg(value):
        lista += 1
        suma += value
        return suma/lista

    return running_avg

avg = average()
```

In [ ]:
```
avg(10)
```

Nos lo echamos otra vez...

Tenemos que usar `nonlocal`

In [ ]:
```
def average():
    suma = 0.
    lista = 0
    def running_avg(value):
        nonlocal suma, lista
        lista += 1
        suma += value
        return suma/lista

    return running_avg

avg = average()
```

In [ ]:
```
print(avg(1))
print(avg(5))
print(avg(10))
print(avg(3))
print(avg(50))
```

In [ ]:
```
print(avg.__code__.co_freevars)
print(avg.__closure__)
```

`@classmethod`

- Metodos asociados a una clase
- Distinto a un metodo estatico

In [ ]:
```
def m_clase(cls): pass
def m_static(): pass
def m_instance(self): pass
```

- Metodos de instancia: reciben `self` como argumento en la primera posicion. Reciben la instancia
- Metodos de clase: reciben `cls` como argumento en la primera posicion. Reciben la clase
- Metodos estaticos: no reciben algo obligados en la primera posicion. **NO** tienen acceso ni a la clase ni la instancia

Un ejemplo

In [ ]:
```
%reset -f
```

In [ ]:
```
class A:
    def __init__(self, var1, var2):
        self.var1 = var1
        self.var2 = var2

    def instance_method(self, *args):
        print(self)
        return self

    @classmethod
    def class_method(cls, *args):
        print(cls)
        return cls(*args)

    @staticmethod
    def static_method(*args):
        return A(*args)
```

In [ ]:
```
a = A(1,2)

a.instance_method(1, 2)
print(a.class_method(1, 2))
print(a.static_method(1, 2))
```

La diferencia entonces

In [ ]:
```
class B(A): pass
```

In [ ]:
```
b = B(1,2)

b.instance_method(1, 2)
print(b.class_method(1, 2))
print(b.static_method(1, 2))
```

Usos reales

- `dict.fromkeys(iterable, default_value)`

In [ ]:
```
d_1 = {
    "a": 0,
    "b": 0,
    "c": 0,
    "d": 0,
    "e": 0,
}

print(d_1)
```

In [ ]:
```
d_2 = dict.fromkeys("abcde", 0)

print(d_2)
```

Modificaciones en *Runtime*

- Podemos agregar variables y metodos en tiempo de ejecucion
- Todo es un bloque ejecutable en Python
- Podemos acceder a virtualmente todo desde `__dict__`

In [ ]:
```
%reset -f
```

Agregar variables

In [ ]:
```
class A:
    pass
```

In [ ]:
```
a = A()

try:
    a.variable
except AttributeError as e:
    print(e)
```

In [ ]:
```
a.variable = 150
print(a.variable)
```

Agregar funciones

In [ ]:
```
class A:
    pass

def f():
    print(10)
```

In [ ]:
```
a = A()

try:
    a.f()
except AttributeError as e:
    print(e)
```

In [ ]:
```
a.f = f
a.f()
```

Agregar metodos

In [ ]:
```
class A:
    pass

def f(self):
    print(self)
```

In [ ]:
```
a = A()
a.f = f

try:
    a.f()
except TypeError as e:
    print(e)
```

In [ ]:
```
a.f(100)
```

La buena forma

In [ ]:
```
import types
a.f = types.MethodType(f,  a)
a.f()
```

Decoradores

- Azucar sintactica a aplicacion de funciones
- Recordar que funciones en Python son de primer orden
- Funciones recibiendo funcion de argumento y retornando funciones

In [ ]:
```
%reset -f
```

Motivacion

In [ ]:
```
def print_function_name(function):
    print(function.__name__)
    return function

def f():
    pass
```

In [ ]:
```
f = print_function_name(f)
f()
```

Funciona una pura vez

In [ ]:
```
def print_function_name(function):
    def wrapper():
        print(function.__name__)
        return function
    return wrapper

def f():
    pass
```

In [ ]:
```
f = print_function_name(f)
f()
f()
f()
```

El decorador

In [ ]:
```
@print_function_name
def mi_super_funcion():
    pass
```

In [ ]:
```
mi_super_funcion()
mi_super_funcion()
mi_super_funcion()
```

Decoradores Simples

- Como recibir argumentos
- Como mantener documentacion
- Decorador con argumentos
- Soportar argumentos de decorador, funcion y flexibilidad

- Como recibir argumentos

In [ ]:
```
def print_args(func):
    def wrapper(a):
        print(func.__name__, a)
        return func(a)
    return wrapper

@print_args
def my_name(a):
    return a
```

In [ ]:
```
my_name(100)
```

La buena forma

In [ ]:
```
def print_args(func):
    def wrapper(*args, **kwargs):
        print(func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

@print_args
def my_name(a):
    return a

my_name(100)
```

- Como mantener documentacion

In [ ]:
```
import functools
```

In [ ]:
```
def print_args(func):
    ################
    @functools.wraps(func)
    ################
    def wrapper(*args, **kwargs):
        print(func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper
```

- Decorador con argumentos

In [ ]:
```
def caller(num=10):
    def inner_decorator(func):
        print(f"Inside inner_decorator with func: {func.__name__}")
        @functools.wraps(func)
        def decorator(*args, **kwargs):
            print(f"Inside decorator with args: {args} {kwargs}")
            return func(*args, **kwargs)
        return decorator

    print(f"Inside caller with arg: {num}")
    return inner_decorator

@caller(num=100)
def f(arg1, arg2):
    print(arg1, arg2)
```

In [ ]:
```
f(1,2)
```

- Soportar argumentos de decorador, funcion y flexibilidad

In [ ]:
```
def caller(_func=None, *, num=10):
    def inner_decorator(func):
        print(f"Inside inner_decorator with func: {func.__name__}")
        @functools.wraps(func)
        def decorator(*args, **kwargs):
            print(f"Inside decorator with args: {args} {kwargs}")
            return func(*args, **kwargs)
        return decorator

    ########################
    if _func is None:
        print(f"With arguments: {num}")
        return inner_decorator
    else:
        print(f"No arguments, falling to default decorator")
        return inner_decorator(_func)

print("F1------")
@caller(num=100)
def f1(arg1, arg2):
    print(arg1, arg2)

print("F2------")
@caller
def f2(a,b,arg1=3, arg2=5):
    print(a, b, arg1, arg2)
```

In [ ]:
```
print("F1------")
f1(1,2)
print("F2------")
f2(10,20)
```

Guardar informacion en funciones

- Como las funciones tambien son objetos, podemos agregarle variables

In [ ]:
```
def counter(func):
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return func(*args, **kwargs)

    wrapper.calls = 0
    return wrapper

@counter
def fun(a):
    print(a)
```

In [ ]:
```
for i in range(10):
    fun(i)

fun(100)
fun(150)

print(f"Num calls: {fun.calls}")
```

Decoradores de clases

- Tambien podemos decorar clases
- No estamos obligados a retornar cosas distintas siempre
- Podemos recibir la clase, modificarla, y devolverla

In [ ]:
```
def decorator(cls):
    print(cls)
    cls.variable1 = 10
    # podria ser una funcion con nombre tambien
    setattr(cls, "funcion", lambda self, x: self.variable1 + x)
    return cls

@decorator
class A: pass
```

In [ ]:
```
a = A()
print(a.variable1)
print(a.funcion(100))
```

Decoradores como clases

- Podemos simular una clase como una funcion

In [ ]:
```
class Counter:
    def __init__(self, func):
        self.calls = 0
        self.func = func
        functools.update_wrapper(self, func)
    def __call__(self, *args, **kwargs):
        self.calls += 1
        return self.func(*args, **kwargs)

@Counter
def fun(a):
    print(a)
```

In [ ]:
```
for i in range(10):
    fun(i)

fun(100)
fun(150)

print(f"Num calls: {fun.calls}")
```

Ejemplos

Singleton

- Tener maximo una instancia de una clase

In [ ]:
```
def singleton(cls):
    @functools.wraps(cls)
    def wrapper(*args, **kwargs):
        if not wrapper.instance:
            wrapper.instance = cls(*args, **kwargs)
        return wrapper.instance
    wrapper.instance = None
    return wrapper

class A:
    pass

@singleton
class B:
    pass
```

In [ ]:
```
a1 = A()
a2 = A()

b1 = B()
b2 = B()

print(a1 is a2)
print(b1 is b2)
```

Cache

- Guardar ejecuciones anteriores para no recalcular

In [ ]:
```
@functools.lru_cache(maxsize=20)
def recursive(num):
    print(num)
    if num <= 0:
        return num
    return recursive(num-1)
```

In [ ]:
```
for i in range(10):
    recursive(i)

for _ in range(100):
    for i in range(4):
        recursive(i)

print(recursive.cache_info())
```

Mas en [https://wiki.python.org/moin/PythonDecoratorLibrary (https://wiki.python.org/moin/PythonDecoratorLibrary)](https://wiki.python.org/moin/PythonDecoratorLibrary)

Propiedades (`property`)

- Crear accesores personalizados para variables
- Accesores a composiciones de variables o funciones

In [ ]:
```
%reset -f
```

Usando una funcion

In [ ]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def get_full_name(self):
        return f"{self.name} {self.surname}"

    def set_full_name(self, full_name):
        self.name, self.surname = full_name.split()
```

In [ ]:
```
person = Person("JP", "Silva")
print(person.get_full_name())

person.set_full_name("J-P Silva")

print(person.get_full_name())
```

Pero que lata poner todo el rato `getVAR`, `setVAR` ...

In [ ]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def get_full_name(self):
        return f"{self.name} {self.surname}"

    def set_full_name(self, full_name):
        self.name, self.surname = full_name.split()

    full_name = property(get_full_name, set_full_name)
```

In [ ]:
```
person = Person("JP", "Silva")
print(person.full_name)

person.full_name = "J-P Silva"

print(person.full_name)
```

Como decorador

In [ ]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def full_name(self):
        return f"{self.name} {self.surname}"

    @full_name.setter
    def full_name(self, person_name):
        self.name, self.surname = person_name.split()
```

In [ ]:
```
person = Person("JP", "Silva")
print(person.full_name)

person.full_name = "J-P Silva"

print(person.full_name)
```

Descriptores

- Mas flexibles
- Bajo nivel
- Especificos cuando realmente necesitamos algo

In [ ]:
```
class NumberProxy:
    def __init__(self, value):
        self.value = value

    def __get__(self, obj, objtype):
        return self.value

class Container:
    a = NumberProxy(100)

c = Container()
c.a
```

Dunder methods / Magic Methods

- *Operator overloading* es otro nombre
- Son bacanes
- Son utiles
- Usenlos

In [ ]:
```
%reset -f
```

Implementemos una lista de Anime (such weeb)

- Una clase `Anime`
- Una clase `AnimeList`

Nivel 1

- Anime

In [ ]:
```
_id_count = 0

class Anime:
    def __init__(self, name, ranking):
        global _id_count
        self.id = _id_count
        _id_count += 1

        self.name=name
        self.ranking = ranking

    def is_higher(self, other_anime):
        return self.ranking > other_anime.ranking
```

In [ ]:
```
all_anime = []
for rank, name in enumerate(["FMA", "Steins;Gate", "HxH", "Kimi no Na Wa"]):
    all_anime.append(Anime(name=name, ranking=rank))
```

- AnimeList

In [ ]:
```
class AnimeList:
    def __init__(self, anime_list=None):
        if anime_list is not None:
            self.anime_list = anime_list
        else:
            self.anime_list = []

    def add_anime(self, anime):
        self.anime_list.append(anime)

    def get_all_anime(self):
        return self.anime_list

    def get_anime(self, indices):
        return self.anime_list[indices]

    def num_anime(self):
        return len(self.anime_list)
```

In [ ]:
```
anime_list = AnimeList()
for anime in all_anime[1:]:
    anime_list.add_anime(anime)
```

In [ ]:
```
print(anime_list.get_all_anime())
print(anime_list.get_anime(2))
print(anime_list.num_anime())
```

Nivel 2

- Anime
    - Contador interno
    - Operator Overloading

In [ ]:
```
class Anime:
    class Ids:
        counter = 0
        def __call__(self):
            self.counter += 1
            return self.counter
    id_generator = Ids()

    def __init__(self, name, ranking):
        self._id = self.id_generator()
        self.name=name
        self.ranking = ranking

    def __eq__(self, other_anime):
        return self.name == other_anime.name and self.ranking == other_anime.rankin
g

    def __repr__(self):
        return f"(id={self._id}, name={self.name}, rank={self.ranking})"

    def __str__(self):
        return self.name
```

In [ ]:
```
all_anime = []
for rank, name in enumerate(["FMA", "Steins;Gate", "HxH", "Kimi no Na Wa"]):
    all_anime.append(Anime(name=name, ranking=rank))
```

- AnimeList
    - Operator Overloading
    - Agregamos Anime al principio y final de la lista

In [ ]:
```
class AnimeList:
    def __init__(self, anime_list=None):
        if anime_list is not None:
            self.anime_list = anime_list
        else:
            self.anime_list = []

    def __add__(self, anime):
        temp = self.anime_list.copy()
        temp.append(anime)
        return AnimeList(temp)

    def __radd__(self, anime):
        temp = self.anime_list.copy()
        temp.insert(0, anime)
        return AnimeList(temp)

    def __eq__(self, other_list):
        return self.anime_list == other_list.anime_list

    def __getitem__(self, indices):
        return self.anime_list[indices]

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

    def __repr__(self):
        return str(self.anime_list)
```

In [ ]:
```
anime_list = AnimeList()
for anime in all_anime[1:]:
    anime_list += anime
```

In [ ]:

```
print(anime_list)
print(anime_list[:-1])
print(len(anime_list))

anime_list += all_anime[0]
anime_list = all_anime[0] + anime_list

print(anime_list)
```

Overkill

- MAS
- iterador

- Anime

In [ ]:

```
from functools import total_ordering
import itertools

@total_ordering
class Anime:
    class NewId:
        gen = itertools.count()
        def __call__(self):
            return next(self.gen)
    new_id = NewId()

    def __init__(self, name, ranking):
        self._id = Anime.new_id()
        self.name=name
        self.ranking = ranking

    def __eq__(self, other_anime):
        return self.name == other_anime.name and self.ranking == other_anime.rankin
g

    def __hash__(self):
        return hash(str(self._id) + self.name)

    def __lt__(self, other_anime):
        return self.ranking < other_anime.ranking

    def __repr__(self):
        return f"(id={self._id}, name={self.name}, rank={self.ranking})"

    def __str__(self):
        return self.name

    def __iter__(self):
        return iter((self.name, self.ranking))
```

In [ ]:

```
all_anime = []
for rank, name in enumerate(["FMA", "Steins;Gate", "HxH", "Kimi no Na Wa"]):
    all_anime.append(Anime(name=name, ranking=rank))
```

- AnimeList

In [ ]:

```
class AnimeList:
    def __init__(self, anime_list=None):
        if anime_list is not None:
            self.anime_list = anime_list
        else:
            self.anime_list = []

    def __add__(self, anime):
        temp = self.anime_list.copy()
        temp.append(anime)
        return AnimeList(temp)

    def __radd__(self, anime):
        temp = self.anime_list.copy()
        temp.insert(0, anime)
        return AnimeList(temp)

    def __eq__(self, other_list):
        return self.anime_list == other_list.anime_list

    def __getitem__(self, indices):
        return self.anime_list[indices]

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

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

    def __contains__(self, anime):
        return anime in self.anime_list

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

In [ ]:

```
anime_list = AnimeList()
for anime in all_anime[1:]:
    anime_list += anime
```

In [ ]:

```
print(anime_list)
print(anime_list[:-1])
print(len(anime_list))

anime_list += all_anime[0]
anime_list = all_anime[0] + anime_list

print(anime_list)

for anime_name, anime_ranking in anime_list:
    print(anime_name, anime_ranking)
```

NotImplemented

In [ ]:
```
class A:
    def __init__(self, var):
        self.var=var

    def __eq__(self, other):
        print("__eq__ not implemented in A")
        return NotImplemented

    def __lt__(self, other):
        print("__lt__ not implemented in A")
        return NotImplemented

class B:
    def __init__(self, var):
        self.var=var

    def __eq__(self, other):
        print("__eq__ is implemented in B")
        return self.var == other.var

    def __gt__(self, other):
        print("__gt__ is implemented in B")
        return self.var > other.var
```

In [ ]:
```
a = A(10)
b = B(1)
print(a==b)
```

In [ ]:
```
b = B(10)
print(a==b)

print("-"*20)

print(a < b)
```

Herencia Multiple

- Miedos de problema del diamante
- Confusion sobre orden de llamado
- Quien es quien

In [ ]:
```
%reset -f
```

Normal

- La herencia que todos hemos visto

In [ ]:
```
class A:
    def __init__(self, var):
        self.var = var
    def mA(self):
        print("Im in A")

class B(A):
    def __init__(self, var1, var2):
        super().__init__(var1)
        self.var2 = var2

    def mB(self):
        print("Im in B")
```

In [ ]:
```
a = A(10)
b = B(100, 200)

print("a is A", isinstance(a, A))
print("a is B", isinstance(a, B))

print("b is A", isinstance(b, A))
print("b is B", isinstance(b, B))

print("Metodos")
b.mA()
b.mB()

print("b variables", b.var, b.var2)
```

Eleccion de super

- No lo usen
- En serio
- Bueno si, pero si lo usan mucho tienen un problema de abstraccion

In [ ]:

```
class A:
    def m(self):
        print("Im in A")

class B(A):
    def m(self):
        print("Im in B")

class C(B):
    def m(self):
        print("Im in C")
```

In [ ]:
```
class D(C):
    def m(self):
        print("Im in D")

    def callerD(self):
        self.m()
    def callerC(self):
        super(D, self).m()
    def callerB(self):
        super(C, self).m()
    def callerA(self):
        super(B, self).m()
```

In [ ]:
```
d = D()
d.callerD()
d.callerC()
d.callerB()
d.callerA()
```

Problemas con herencia multiple

In [ ]:
```
class Student:
    def __init__(self, student_id, school, *args, **kwargs):
        self.student_id = student_id
        self.school = school

    def identification(self):
        return f"ID:{self.student_id}-{self.surname}-{self.school}"

class Human:
    def __init__(self, name, age, *args, **kwargs):
        self.name = name
        self.age = age

    def identification(self):
        return f"{self.name}"

class Yo(Student, Human):
    pass
```

In [ ]:
```
yo = Yo(0, "JP Silva", "UChile", 500)
print(yo.identification())
```

Method Resolution Order (MRO)

In [ ]:
```
Yo.__mro__
```

In [ ]:
```
class Yo(Human, Student):
    pass
```

In [ ]:
```
yo = Yo(0, "JP Silva", "UChile", 500)
print(yo.identification())
```

??????????????????

Probemos otra cosa

In [ ]:
```
class Yo(Human, Student):
    def identification(self):
        return Human.identification(self)

    def school_identification(self):
        return Student.identification(self)
```

In [ ]:
```
yo = Yo(student_id=0, name="JP Silva", school="UChile", age=500)
print(yo.identification())
print(yo.school_identification())
```

oof que pasa

La buena forma, inicializar a todos los padres manualmente

In [ ]:
```
class Yo(Human, Student):
    def __init__(self, name, *args, **kwargs):
        Human.__init__(self, *args, name=name, **kwargs)
        Student.__init__(self, *args, name=name, **kwargs)

        self.surname = name.split()[1]

    def identification(self):
        return Human.identification(self)

    def school_identification(self):
        return Student.identification(self)
```

In [ ]:
```
yo = Yo(student_id=0, name="JP Silva", school="UChile", age=500)
print(yo.identification())
print(yo.school_identification())
```

Mixins

- Clases pequeñas que definen funcionalidad
- Como un template, para los que pasaron metodologias

In [ ]:
```
class Mixin:
    def mixin(self, arg1, arg2):
        print(arg1, arg2)

        # Mixin no tiene `var`
        return self.var

class SuperClass:
    def m(self):
        print("In SuperClass")

class Example(SuperClass, Mixin):
    def __init__(self, var):
        self.var = var
```

In [ ]:
```
e = Example(5)
print(e.mixin(1, 2))
```

Mixin vs Clases abstractas?

- Mixin provee de funcionalidad pero no puede usarla directamente
- Clase abstracta provee de una interfaz. **No** tiene funcionalidad, el usuario debe implementarla

- `sklearn` esta ***lleno*** de Mixins: https://github.com/scikit-learn/scikit-learn/blob/1495f6924/sklearn/base.py#L328 (https://github.com/scikit-learn/scikit-learn/blob/1495f6924/sklearn/base.py#L328)

Metaclases

Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why)\ -- Tim Peters

- 99% del tiempo no necesitaran modificar la inicializacion de una clase.
- Del 1% cuando si lo necesitas, hay otras formas como decoradores de clases y especificaciones en __new__, esto es el 99% de las veces cuando si necesitas modificar una clase.
- El 1% del 1% de las veces, necesitas metaclases. No es un *feature* que sea usable para el publico general, pero si no sabes que existe, y no sabes para que sirve, nunca podras saber si en algun momento lo necesitas o no.

In [ ]:
```
%reset -f
```

Objeto como instancia de una clase

In [ ]:
```
class A:
    def __init__(self, var):
        self.var = var

a = A("hola")

print(a.var)
```

In [ ]:
```
a.v1 = 10
print(a.v1)
```

In [ ]:
```
def example(arg):
    print(arg)

example(a)
```

Las clases tambien son objetos

Hmmmmm...

Clases como objetos

In [ ]:
```
class A:
    def __init__(self, var):
        self.var = var
```

In [ ]:
```
print(A)
```

In [ ]:
```
A.something = 100

print(A.something)
```

In [ ]:
```
otherA = A
print(otherA.something)
```

In [ ]:
```
print(otherA("Otra clase"))
```

Muy util

In [ ]:
```
class_list = []
for i in range(5):
    ###### Clase dentro? ######
    class AClass:
        var = i
        def __init__(self, arg):
            print(arg)

    class_list.append(AClass)
```

In [ ]:
```
print(class_list)
print([c.var for c in class_list])
```

In [ ]:
```
class_instances = []
for c, arg in zip(class_list, "holaquetal"):
    class_instances.append(c(arg))

print(class_instances)
```

Quien eres?

In [ ]:
```
a = A("holi")
print(a)
print(a.__class__)
```

In [ ]:
```
print(A)
print(A.__class__)
```

In [ ]:
```
print(a.__class__.__class__)
```

`type` como funcion y como clase?

Como funcion

- Dame un objeto y te digo quien es

In [ ]:
```
print(type(a))
```

In [ ]:
```
print(type(10))
```

In [ ]:
```
print(type("Hola"))
```

In [ ]:
```
print(type(True))
```

In [ ]:
```
print(type(type(True)))
```

Como funcion v2

- Dame varias cosas y te doy una clase
- ??????????????

type(class, bases, dict)

In [ ]:
```
B = type("B", (), {})
```

In [ ]:
```
print(B)
print(B())
```

Con variables

In [ ]:
```
C = type("C", (B, A), {"a":1, "b":2})
print(C)
```

In [ ]:
```
c = C(100)
print(c.__dict__)
```

In [ ]:
```
variables = []
for member_name in dir(c):
    if "__" not in member_name:
        value = getattr(c, member_name)
        variables.append(f"{member_name}={value}")
print(", ".join(variables))
```

Con metodos

In [ ]:
```
def m(self, var):
    print(self.a, var)
D = type("D", (), {"a":1, "m":m})
```

In [ ]:
```
d = D()
d.m(100)
```

Metaclases

- Una clase es una fabrica de objetos
- Una metaclase es una fabrica de clases
- `type` es una metaclase
- `type` es super clase de si mismo

In [ ]:
```
d = D()
print(d.__class__)
print(d.__class__.__class__)
print(d.__class__.__class__.__class__)
dir(d.__class__.__class__)
```

Mi primera metaclase

- pero es una funcion (?)

In [ ]:
```
def all_dunder(cls, bases, attrs):
    dunder_attrs = {}
    for name, val in attrs.items():
        if not name.startswith('__'):
            print(f"Replacing {name} with __{name}__")
            dunder_attrs[f"__{name}__"] = val
        else:
            dunder_attrs[name] = val
    return type(cls, bases, dunder_attrs)
```

In [ ]:
```
class AllDunder(metaclass=all_dunder):
    def a(arg):
        return arg

    def b(arg):
        return arg

    def c(arg):
        return arg
```

In [ ]:
```
dunder = AllDunder()
dir(dunder)
```

Como clase ahora

In [ ]:
```
class AllDunder(type):
    def __new__(cls, clsname, bases, attrs, **kwargs):
        print(kwargs)

        dunder_attrs = {}
        for name, val in attrs.items():
            if not name.startswith('__'):
                print(f"Replacing {name} with __{name}__")
                dunder_attrs[f"__{name}__"] = val
            else:
                dunder_attrs[name] = val
        return super().__new__(cls, clsname, bases, dunder_attrs)
```

In [ ]:
```
class AllDunder(metaclass=AllDunder, arg1="do this", arg2="do that"):
    def a(arg):
        return arg

    def b(arg):
        return arg

    def c(arg):
        return arg
```

Como funciona Django

- Con metaclases

In [ ]:
```
class IntContainer:
    ...
    pass

class StrContainer:
    ...
    pass
```

In [ ]:

```
class FixFields(type):
    def fix_complicated_database_logic(new_class, var_name, var_value):
        ...
        def get_prop(self):
            return getattr(self, "__" + var_name)
        def set_prop(self, value):
            return setattr(self, "__" + var_name, value)

        # este None es la logica complicada de meterse a la BD
        setattr(new_class, "__" + var_name, None)
        setattr(new_class, var_name, property(get_prop, set_prop))

    def __new__(cls, clsname, bases, attrs, **kwargs):
        filter_prop = {}
        std_dict = {}
        for name, val in attrs.items():
            if not name.startswith('__'):
                filter_prop[name] = val
            else:
                std_dict[name] = val

        new_cls = super().__new__(cls, clsname, bases, std_dict)
        setattr(new_cls, "attributes", set())

        for name, val in filter_prop.items():
            if not name.startswith('__'):
                FixFields.fix_complicated_database_logic(new_cls, name, val)

                getattr(new_cls, "attributes").add(name)

        setattr(new_cls, "attributes", frozenset(getattr(new_cls, "attributes")))
        return new_cls
```

In [ ]:
```
class Model(metaclass=FixFields):
    def __init__(self, **kwargs):
        for arg, val in kwargs.items():
            setattr(self, arg, val)
```

In [ ]:
```
class ComplicatedAttrs(Model):
    age = IntContainer()
    name = StrContainer()
```

In [ ]:
```
db = ComplicatedAttrs(age=10, name="Hola")
print(db.attributes)
```

In [ ]:
```
print(f"age = {db.age}")
print(f"name = {db.name}")
```

In [ ]:
```
db.age=100
print(f"age = {db.age}")
```

Virtual Super Classes

- Clases que nunca extendi, pero de alguna manera si lo hice
- Padres no reconocidos

In [ ]:
```
%reset -f
```

In [ ]:
```
from collections.abc import Sized
```

In [ ]:
```
class A:
    def __len__(self):
        return 10
```

In [ ]:
```
a = A()
print(isinstance(a, Sized))
```

??????????????????????????

Abstract Meta Class

- Como si no fuera suficientemente confuso

In [ ]:
```
dir(Sized)
```

In [ ]:
```
print(Sized.__abstractmethods__)
```

In [ ]:
```
import inspect
print(inspect.getsource(Sized.__class__))
```

Otros temas que no vimos

- Context Managers
- Generadores y corutinas
- Async
- Cython y GIL

In [ ]:
```
%reset -f
```

__slots__

In [ ]:
```
class A:
    __slots__ = "a", "b"
    def __init__(self, a, b):
        self.a = a
        self.b = b

class B:
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

In [ ]:
```
from sys import getsizeof

v = A(10, 20)
print(getsizeof(v.__slots__))

vb = B(10, 20)
print(getsizeof(vb.__dict__))
```

WTF Python

In [ ]:
```
%reset -f
```

Reemplazar `__class__`

- Because why not

In [ ]:
```
class Node:
    def __init__(self, function):
        assert hasattr(function, "__call__")
        self.operation = function
        self.num_arguments = function.__code__.co_argcount
        self.arguments = []

    def eval(self):
        return self.operation(*[node.eval() for node in self.arguments])

    def replace(self, otherNode):
        assert isinstance(otherNode, Node)
        self.__class__ = otherNode.__class__
        self.__dict__ = otherNode.__dict__
```

In [ ]:
```
class AddNode(Node):
    num_args = 2
    def __init__(self, left, right):
        super(AddNode, self).__init__(lambda x,y: x+y)
        self.arguments.append(left)
        self.arguments.append(right)
    def __repr__(self):
        return "({} + {})".format(*self.arguments)
```

In [ ]:
```
class TerminalNode(Node):
    num_args = 0
    def __init__(self, value):
        super(TerminalNode, self).__init__(lambda:None)
        self.value = value
    def __repr__(self):
        return str(self.value)
    def eval(self):
        return self.value
```

In [ ]:
```
n = AddNode(TerminalNode(1), TerminalNode(2))

print(f"n: {n}=", n.eval())
print(isinstance(n, AddNode))
```

In [ ]:
```
n.replace(TerminalNode(200))

print(f"n: {n}=", n.eval())
print(isinstance(n, AddNode))
print(isinstance(n, TerminalNode))
```

Parametros mutables default

El error

In [ ]:
```
def f(value, a=[]):
    a.append(value)
    ...
    return sum(a)
```

In [ ]:
```
returns = []
for i in range(10):
    returns.append(f(i))

print(returns)
```

In [ ]:
```
print(f.__defaults__)
```

La solucion

In [ ]:
```
def f(value, a=None):
    if a is None:
        a = []
    a.append(value)
    ...
    return sum(a)
```

In [ ]:
```
returns = []
for i in range(10):
    returns.append(f(i))

print(returns)
```

In [ ]:
```
print(f.__defaults__)
```

257 no es 257?

In [ ]:
```
%reset -f
```

In [ ]:
```
a=1
b=1
print(a is b)

a=257
b=257
print(a is b)
```

Cuidado con las referencias

In [ ]:
```
row = [""]*3
print(row)

board = [row]*3
print(board)
```

In [ ]:
```
board[0][1] = "X"
print(board)
```

Function Closure

In [ ]:
```
def f(n):
    res = []
    for x in range(n):
        def aux():
            return x + 10
        res.append(aux)
    return res
```

In [ ]:
```
partial_fns = f(10)
print(partial_fns)
```

In [ ]:
```
print([fn() for fn in partial_fns])
```

▪▪▪

In [ ]:
```
def f(n):
    res = []
    for x in range(n):
        ##########.###########
        def aux(x=x):
            return x + 10
        res.append(aux)
    return res
```

In [ ]:
```
partial_fns = f(10)
print(partial_fns)
```

In [ ]:
```
print([fn() for fn in partial_fns])
```

Contacto

Gracias!

Juan-Pablo Silva
jpsilva@dcc.uchile.cl
jpsilva.cl