# Ayudantía Examen 1

Ivania Donoso

# Contenidos del curso

- Programación Orientada a Objetos
- Estructuras de Datos
- Funciones de Python y Programación Funcional
- Meta Clases
- Manejo de Excepciones
- Testing
- Simulación
- Threading
- Interfaces Gráficas
- I/O
- Networking
- Webservices

# Contenidos del curso

- Programación Orientada a Objetos
- Estructuras de Datos
- Funciones de Python y Programación Funcional
- Meta Clases
- Manejo de Excepciones
- Testing

- Simulación
- Threading
- Interfaces Gráficas
- I/O
- Networking
- Webservices

# Programación Orientada a Objetos

- Objetos
- Herencia
- Herencia múltiple
- Polimorfismo
- Clases Abstractas
- Properties

¿Cómo practicar?

- Hacer el modelo de las tareas
- Hacer modelos de las actividades desde simulación

# Programación Orientada a Objetos

En un curso de desarrollo de software se arman equipos de k alumnos que trabajan para un cliente. Cada grupo de alumnos es supervisado periódicamente por un ayudante y el profesor del curso.

Cada alumno tiene una probabilidad de encontrarse enojado y una probabilidad de renunciar. Además, un alumno puede programar un cierto número de líneas de código por cada tarea que se le encomienda, que depende del lenguaje:

- Python: Random entre 1 y 600 líneas
- Javascript: Random entre 1 y 400 líneas
- C: Random entre 1 y 200 líneas

Dentro del equipo hay un alumno especial que nunca renuncia, el jefe de grupo. Tanto el profesor como el ayudante pueden realizar comentarios. Cada uno de ellos tiene una probabilidad de decir un comentario positivo (y en caso contrario, un comentario negativo). Cada ayudante tiene una probabilidad de solucionar un conflicto del grupo que supervisa. Mientras tanto, el cliente tiene una probabilidad de máximo un 10% de hacer boicot al proyecto, ya sea porque el resultado no le gustó, o porque es tan bueno que no lo puede entender. Modele esta situación con OOP implementando las clases necesarias en Python.

# Estructuras de Datos

- Árboles
- Diccionarios
- Colas
- Stacks
- Sets

¿Cómo practicar?

- Hacer la actividad de EDD de nuevo (mejor y con bonus)
- Hacer la actividad de EDD del semestre pasado
- Estudiar la diferencia entre getitem, iterador, iterable

# Programación Funcional

- Comprensión de listas
- Iterables e iteradores
- Generadores
- Funciones lambda
- Map
- Reduce
- Filter
- Decoradores

¿Cómo practicar?

- Leer este post http://nvie.com/posts/iterators-vs-generators/
- Hacer los ejercicios de ies anteriores y de las ayudantías
- Transformar código de sus actividades

# Programación Funcional

```
f1 = lambda x: (x+1) ** 2
v1 = [[1, 2, 4], [1, 3], [2, 5, 9], [1, 6]]
v2 = list(map(lambda y: list(filter(lambda x: x > 10, list(map(f1, y))
    )), v1))
print(v2)
```

# Programación Funcional

```
def f2(b, item):
    lc = b[:]
    lc.remove(item)
    return lc

def f1(a):
    if len(a) == 0:
        return [[]]
    return [[x] + y for x in a for y in f1(f2(a, x))]

print(f1(["a","b","c"]))
```

# MetaClases

Comprender la lógica detrás de la construcción y creación de clases

¿Cómo practicar?

- Hacer la actividad de metaclases
- Hacer y entender la tabla de comparación
- Entender la diferencia entre los métodos y atributos de clase y de instancia

| | Clase | MetaClase |
|---|---|---|
| `__new__` | cls, args, kwargs | meta, name, bases, attributes |
| | objeto de Clase cls | clase del tipo *meta*, nombre *name* que hereda de *bases* y que tiene *attributes* atributos |
| `__init__` | self (creado por el new), args, kwargs | cls (creada por el new), name, bases, attributes |
| | nada | clase cls inicializada |
| `__call__` | self, args, kwargs | cls, args, kwargs |
| | no está obligado a retornar | un objeto de la clase cls creado e inicializado con args y kwargs |

```python
class Examen:

    def __new__(cls, *args, **kwargs):
        cls.students_dict = {}
        cls.id_ = cls.generate_user_id()
        return super().__new__(cls)

    def __init__(self, name):
        self.name = name

    def __call__(self, *args, **kwargs):
        return [Examen.students_dict[ar] for ar in args]

    @staticmethod
    def generate_user_id():
        count = 0
        while True:
            yield count
            count += 1

    def add_user(self, name):
        Examen.students_dict[name] = next(Examen.id_)

if __name__ == "__main__":
    e = Examen("Progra")
    e.add_user("E1")
    e.add_user("E2")
    e.add_user("E3")
    print(e.students_dict)
    print(e("E1", "E2", "E3"))
```

# Preguntas

- ¿Qué representa `cls`?
- El método `generate_user_id` ¿pertenece a la clase o a las instancias de `Examen`?
- ¿Qué imprime la sentencia `print(e.students_dict)`?
- ¿Qué imprime la sentencia `print(e("E1", "E2", "E3"))`?

# Cuál es el output?

```
class B:
    pass


class Meta(type):

    def __new__(cls, name, bases, attr):
        print("in 1...")
        print(cls)
        print(name)
        print(bases)
        name = "cambio_" + name
        bases = (B, )
        attr["new"] = 1
        return super().__new__(cls, name, bases, attr)

    def __init__(cls, name, bases, attr):
        print("in 2...")
        print(cls)
        print("name: ", name)
        print("bases: ", bases)
        print("attr: ", attr)
        print(cls.__name__)
        return super().__init__(name, bases, attr)

if __name__ == "__main__":

    class A(metaclass=Meta):
        pass

    print(A.__mro__)
```

# Manejo de Excepciones y Testing

- Manejo de Excepciones:
  - Tipos de excepciones
  - Control de excepciones
- Testing:
  - Tests unitarios en Unittest

¿Cómo practicar?

- Entender los conceptos (capturar excepción, lanzar excepción, test unitario)
- Entender los métodos importantes(try/except/finally/else, setUp, tearDown, ....)