

Capacitación Semillero DISW

Python Clase 3

Docentes: Lautaro Delgado, Idc0295@gmail.com
Magdalena Bouza, bouza.magdalena@gmail.com

Cronograma



Temario

- **Clase 1:**
 - Instalación de Python y herramientas (Anaconda).
 - Preparación del entorno
 - IDE + editor de código
 - Tipos de datos, bucles, funciones
 - OOP en Python
- **Clase 2:**

Librerías: Numpy, Pandas, Matplotlib, Seaborn
- **Clase 3:**

Puesta en producción del código: diseño de una librería/API, haciendo uso de repositorio Git. Unit Testing con PyTest



Python

Clase 3

Índice

- ▷ Buenas Prácticas OOP
- ▷ Introducción a Patrones de Diseño
- ▷ Estructura de un Proyecto en Python
- ▷ Unit Testing
- ▷ Documentación
- ▷ Armado y Publicación de Paquetes
- ▷ Desarrollo de APIs

OOP: Buenas Prácticas

Cohesion

- ▷ Grado en que los elementos de una función o clase pertenecen juntos → *single responsibility principle*

```
def handle_stuff(d: Data,
                quantity: int, screen: int, screen: int,
                status: int, c: Color, ...):
    update_corporate_database(d, q, status)
    for i in range(0, quantity):
        profit[i] = revenue[i] - expense[i] * status
    new_color = c
    status = SUCCESS
    display_profits(screenX, screenY, status, d, c)
```

Coupling

- ▷ Grado de dependencia entre dos porciones de código.

```
def checkEmailSecurity(email):  
    if email.header.bearer.invalid():  
        raise Exception("Email header bearer is invalid")  
    elif email.header.received != email.header.received_spf:  
        raise Exception("Received mismatch")  
    else:  
        print("Email header is secure")
```


SOLID Principles

- S - Single Responsibility
- O - Open/Closed
- L - Liskov Substitution
- I - Interface Segregation
- D - Dependency Inversion

Dependency Inversion

- ▷ Abstraction → Abstract Base Class
- ▷ Types → Type Hints
- ▷ Evitar Coupling

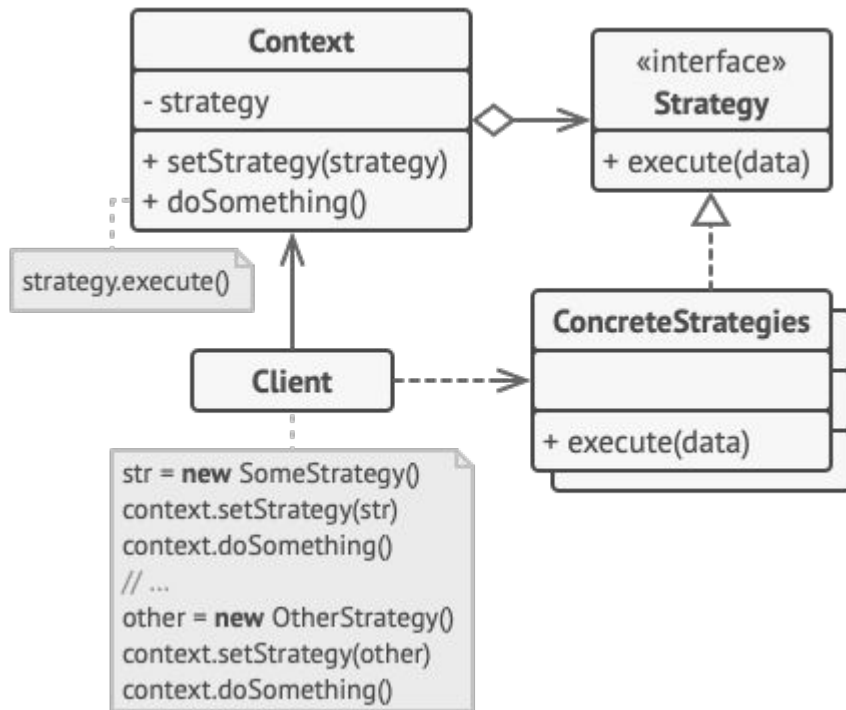
Patrones de Diseño



¿Qué es un patrón de diseño?

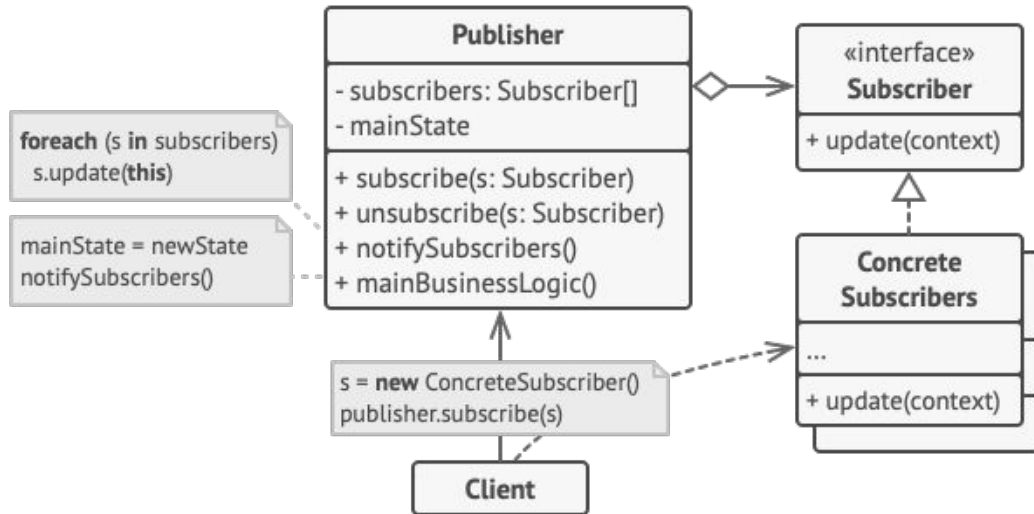
- ▷ Son un juego de herramientas de soluciones comprobadas a problemas habituales en el diseño de software.
- ❑ Creacionales
- ❑ Estructurales
- ❑ Comportamiento

Strategy



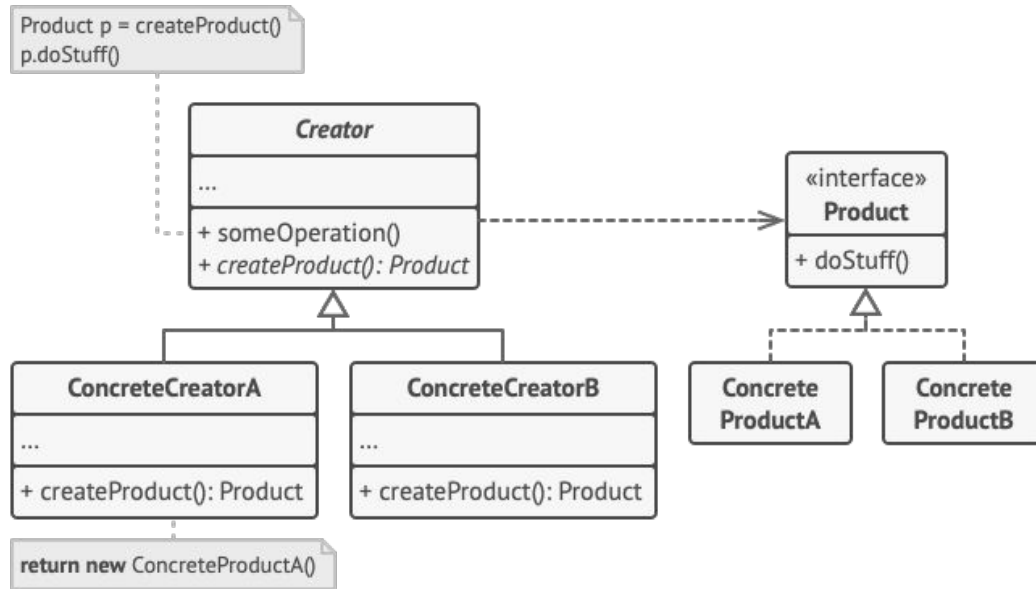
- ▷ “Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.”

Observer



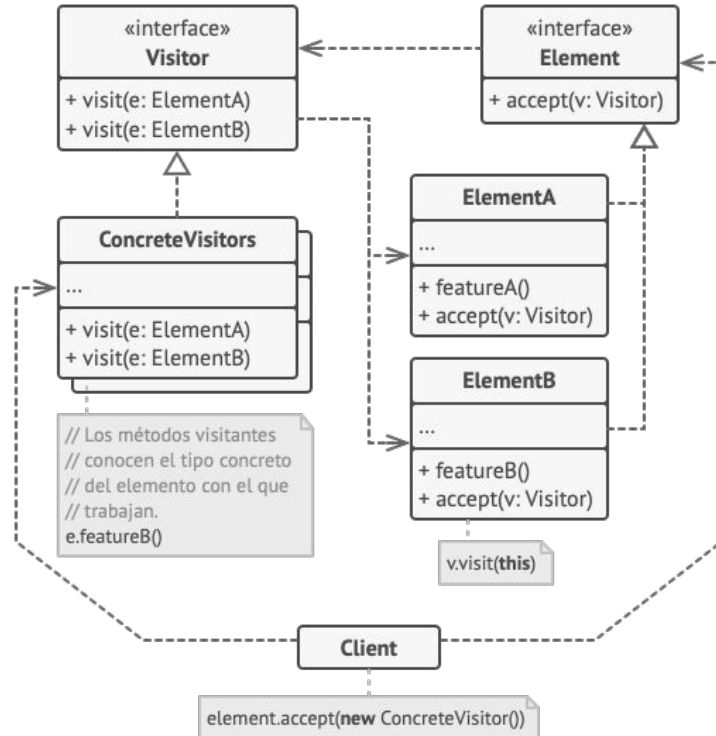
- ▷ “Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.”

Factory



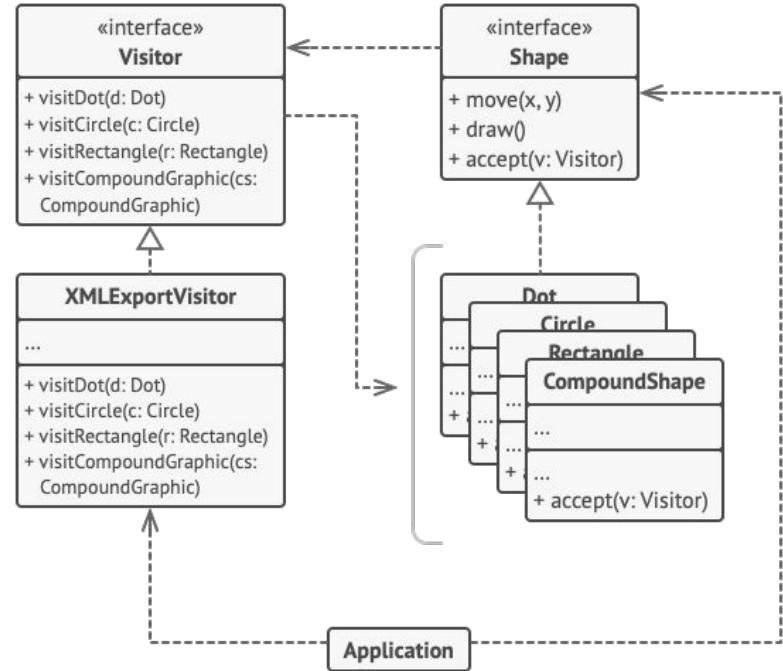
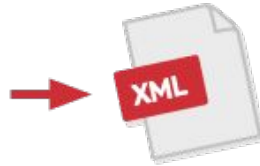
- ▷ “Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclasses alterar el tipo de objetos que se crearán.”

Visitor

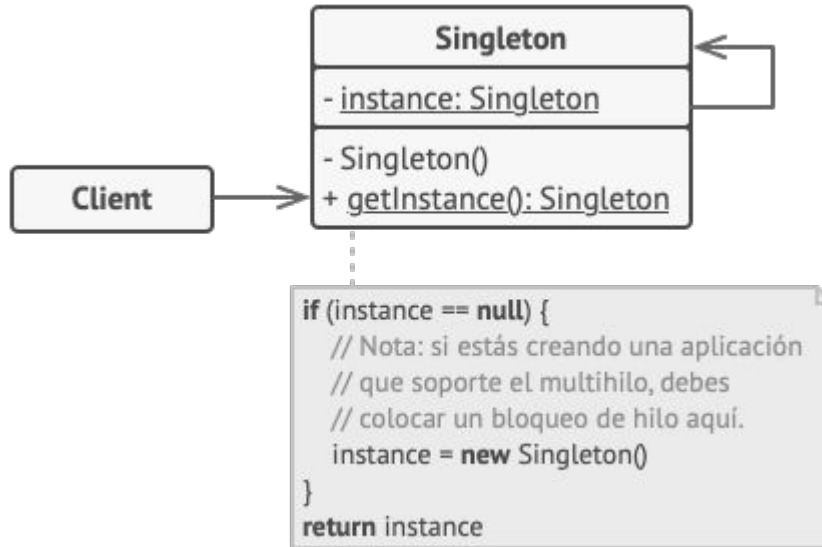


- ▷ “Permite añadir nuevos comportamientos a una jerarquía de clases existente sin alterar el código.”
- ▷ Double dispatch

Visitor

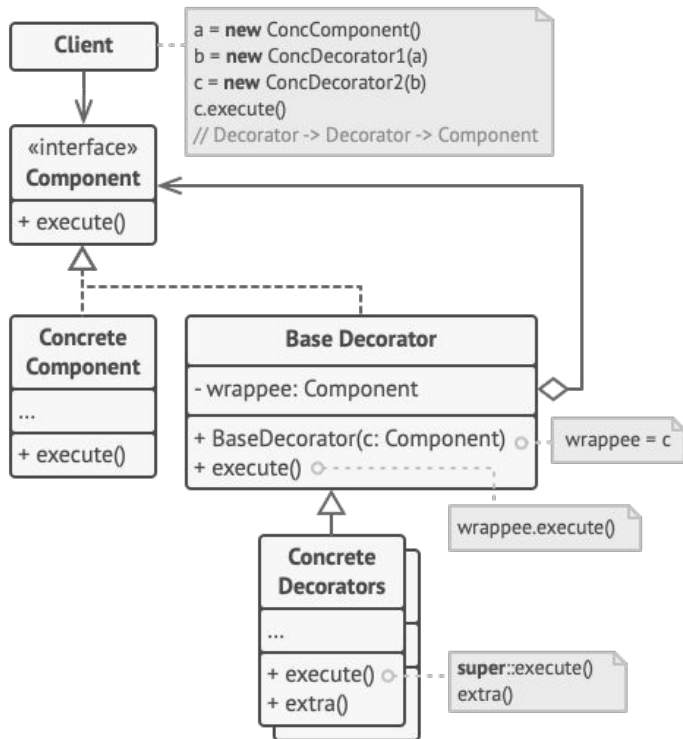


Singleton



- ▷ “Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.”

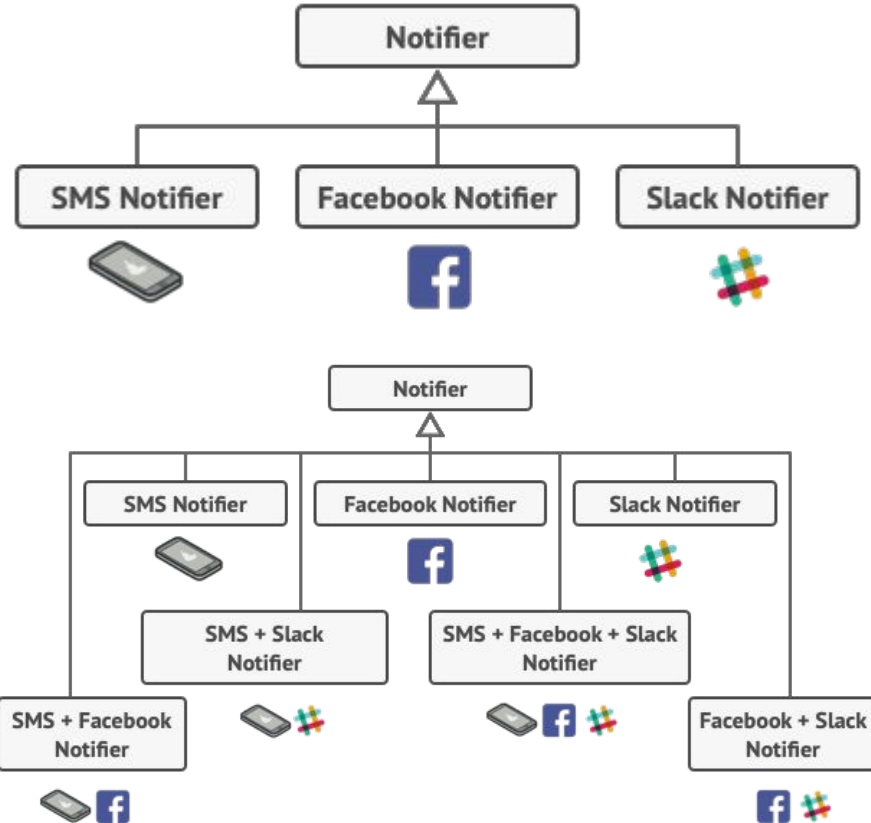
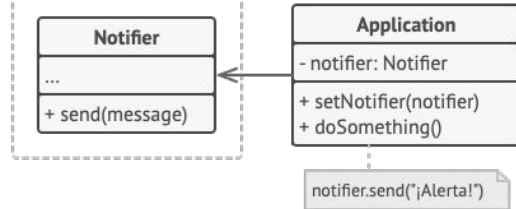
Decorator



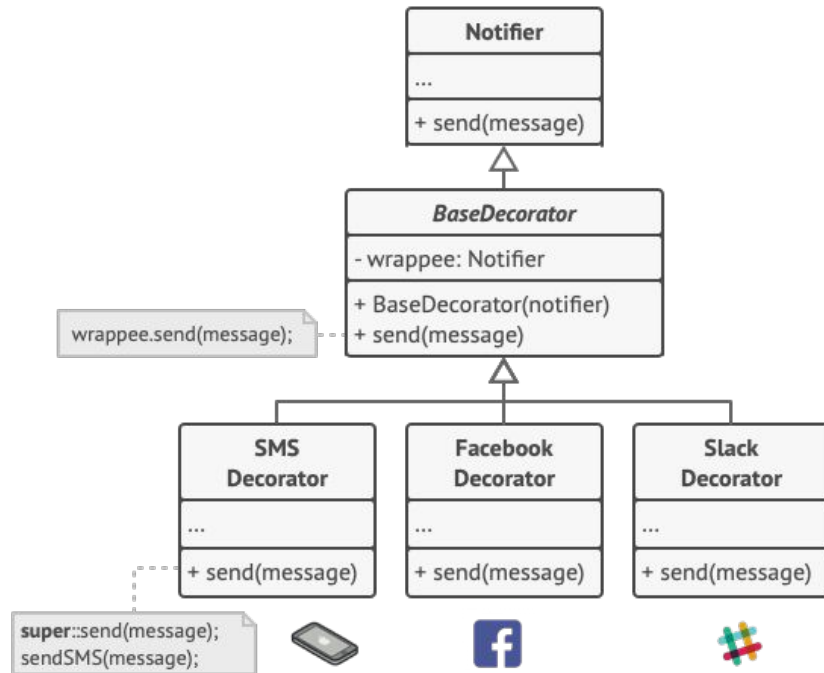
- ▷ “ Permite añadir funcionalidades a objetos mediante encapsulamiento sobre otros”

Decorator

Biblioteca de Notificaciones

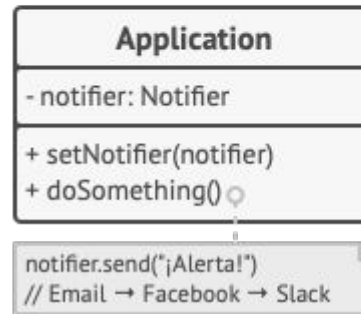


Decorator



```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



Decorator

```
def tracer(func):      @tracer
    def wrapper():      def hello_world():
        print("Entering")    print("Hello World!")
        func()
        print("Exiting")
    return wrapper
```

[Python: How Decorators Function](#)

Estructura de un Proyecto



Módulos

Un módulo permite agrupar clases, funciones y código relacionados. Se lo puede considerar como una librería de código. Puede contener:

- Clases
- Funciones
- Variables
- Código ejecutable
- Atributos asociados

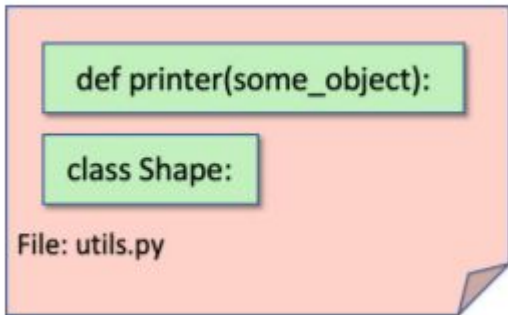
Módulos

Los módulos otorgan:

- ▷ Simplicidad
- ▷ Mantenimiento
- ▷ Testing
- ▷ Reusabilidad
- ▷ Scoping

Módulos

El nombre de un módulo es el nombre del archivo que lo define:



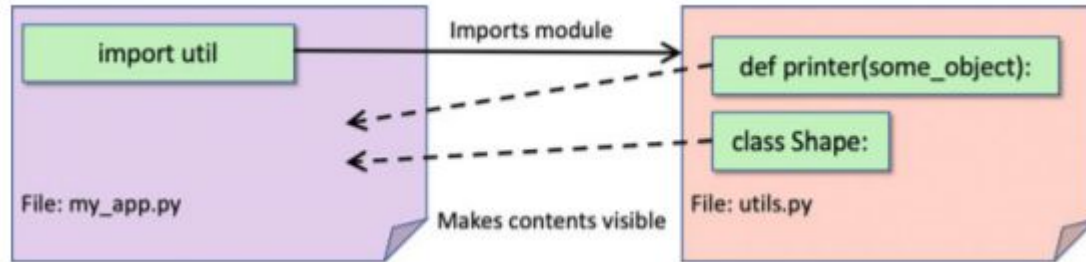
Importar módulos

```
import utils
```

```
utils.printer(utils.default_shape)
```

```
shape = utils.Shape('circle')
```

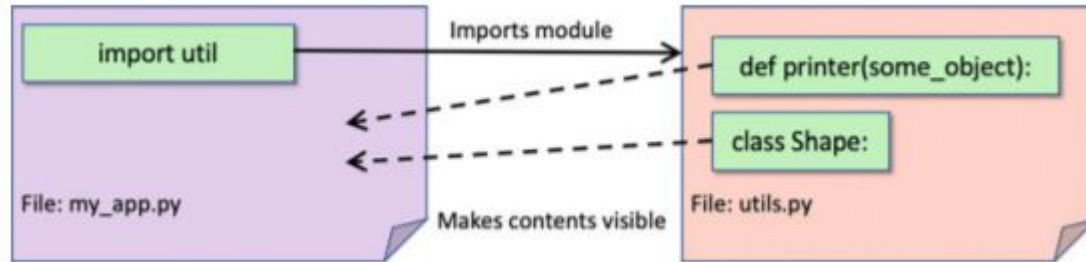
```
utils.printer(shape)
```



Importar desde un módulo

```
from utils import *  
printer(default_shape)  
shape = Shape('circle')  
printer(shape)
```

```
from utils import Shape  
s = Shape('rectangle')  
print(s)
```



Module properties

Generalmente se utilizan para conocer la metadata de un módulo y obtener documentación sobre el mismo

- ▷ `__name__`
- ▷ `__doc__`
- ▷ `__file__`

Module properties

Generalmente se utilizan para conocer la metadata de un módulo y obtener documentación sobre el mismo

- ▷ `__name__`
- ▷ `__doc__`
- ▷ `__file__`

Carga de Módulos

- ▷ Variable de entorno PYTHONPATH
- ▷ `sys.path`
- ▷ orden de búsqueda:
 - directorio actual
 - PYTHONPATH
 - default path (`/usr/local/lib/python/`)

Módulos como scripts

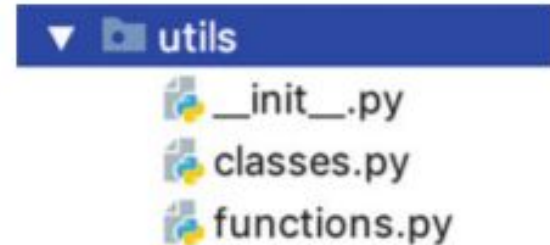
Cualquier módulo es un script y puede ser ejecutado. Se debe distinguir cuando se carga un archivo como módulo o como script standalone.

```
if __name__ == '__main__':
```


Packages

Se pueden organizar distintos módulos en una estructura jerárquica mediante paquetes, haciendo uso de directorios:

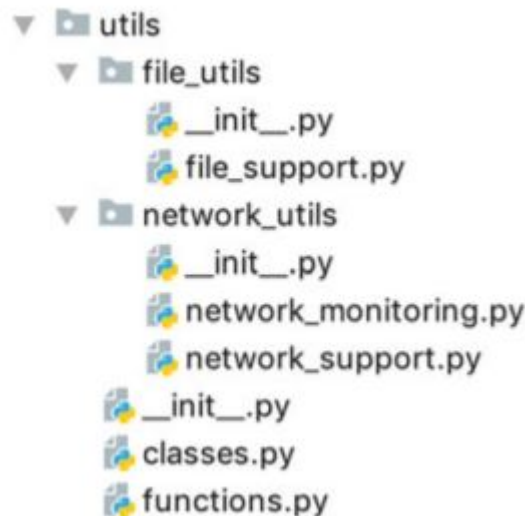
```
from utils.functions import *  
f1()  
  
from utils.classes import *  
p = Processor()
```



Sub Packages

Se pueden organizar estructuras jerárquicas de subpaquetes y módulos dentro de un mismo paquete.

```
from utils.file_utils.file_support
import file_logger
```



Estructura con paquetes internos

```
helloworld/  
├── bin/  
├── docs/  
│   ├── hello.md  
│   └── world.md  
├── helloworld/  
│   ├── __init__.py  
│   ├── runner.py  
│   └── hello/  
│       ├── __init__.py  
│       ├── hello.py  
│       └── helpers.py  
│   └── world/  
│       ├── __init__.py  
│       ├── helpers.py  
│       └── world.py  
├── data/  
│   ├── input.csv  
│   └── output.xlsx  
├── tests/  
│   ├── hello  
│   │   ├── helpers_tests.py  
│   │   └── hello_tests.py  
│   └── world/  
│       ├── helpers_tests.py  
│       └── world_tests.py  
├── .gitignore  
├── LICENSE  
└── README.md
```

- ▷ gitignore
- ▷ [LICENSE](#)
- ▷ [README.md](#)
- ▷ requirements.txt
- ▷ setup.py / setup.cfg

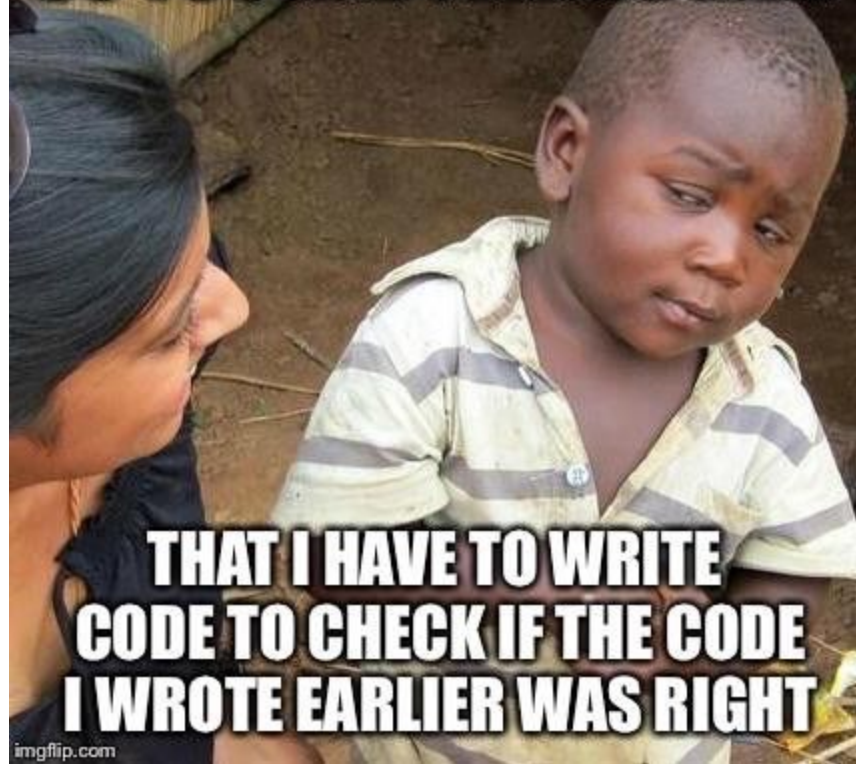
Estructura con paquetes internos

```
helloworld/  
├── bin/  
├── docs/  
│   ├── hello.md  
│   └── world.md  
├── helloworld/  
│   ├── __init__.py  
│   ├── runner.py  
│   └── hello/  
│       ├── __init__.py  
│       ├── hello.py  
│       └── helpers.py  
│   └── world/  
│       ├── __init__.py  
│       ├── helpers.py  
│       └── world.py  
├── data/  
│   ├── input.csv  
│   └── output.xlsx  
├── tests/  
│   ├── hello  
│   │   ├── helpers_tests.py  
│   │   └── hello_tests.py  
│   └── world/  
│       ├── helpers_tests.py  
│       └── world_tests.py  
├── .gitignore  
├── LICENSE  
└── README.md
```

- ▷ bin/
- ▷ docs/
- ▷ {package} /
- ▷ data/
- ▷ test/

Testing

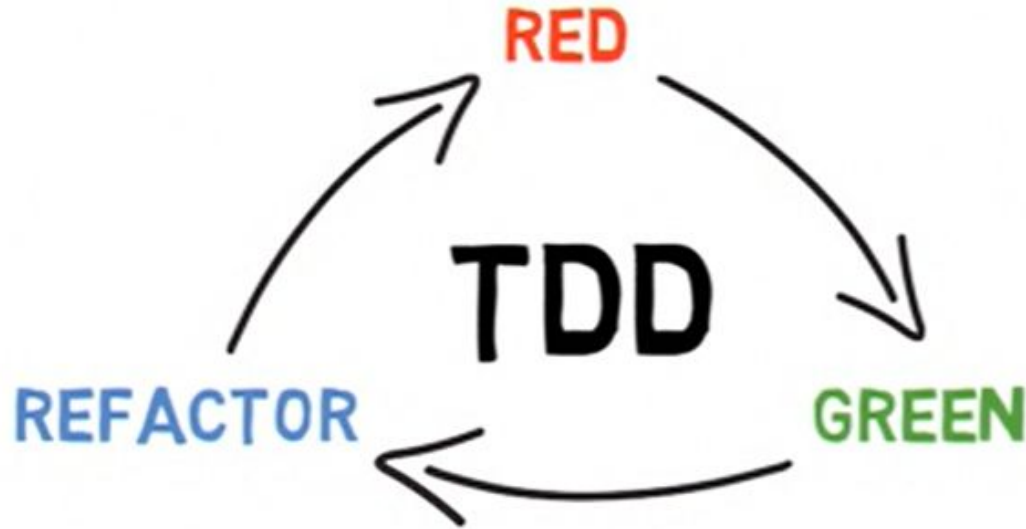
SO YOU ARE TELLING ME...



**THAT I HAVE TO WRITE
CODE TO CHECK IF THE CODE
I WROTE EARLIER WAS RIGHT**

imgflip.com

TDD: Test Driven Development



1. Escribir tests que fallen
2. Correr los tests
3. Escribir el código más simple que pase el test
4. Asegurarse que todos los test pasen (nuevos y viejos)
5. Refactoring

TDD: Test Driven Development

Pros:

- ▷ Fuerza a definir los requerimientos primero
- ▷ Cada requerimiento debe estar bien definido
- ▷ Ahorro de tiempo con detección temprana de errores.
- ▷ Fuerza a escribir código “testable” → patrones de diseño

Contras:

- ▷ Conlleva tiempo agregar todos los tests y mantenerlos.
- ▷ Sentido incorrecto de seguridad (se deben agregar tests de integración, end-to-end, etc)

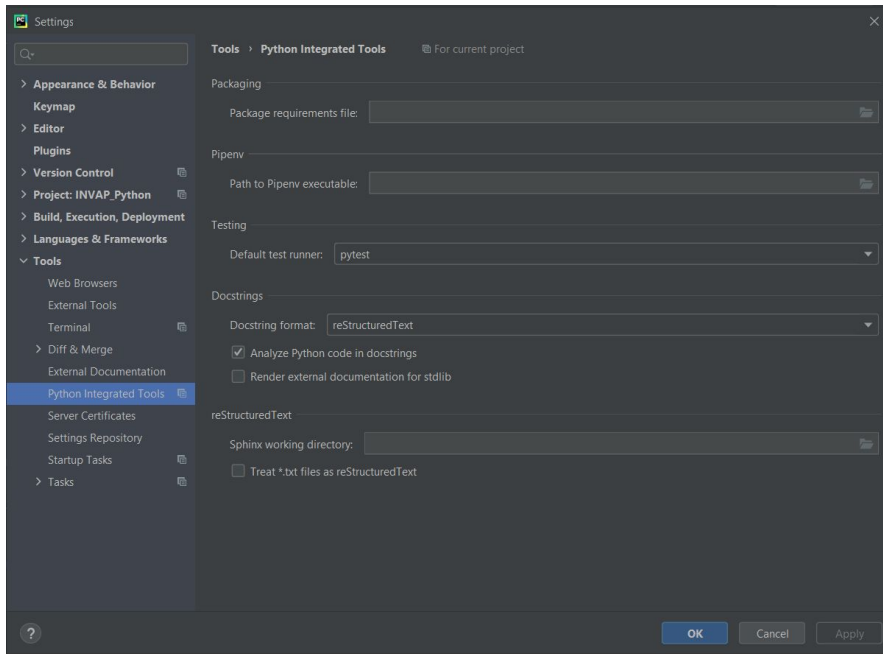
Pytest

- Python unit testing framework.
- Permite crear tests, test modules y test fixtures.
- Usa el assert built-in dentro de Python
- Permite seleccionar por CLI qué tests ejecutar

```
$ pip install pytest
```

Pytest en PyCharm

- Buena integración con PyCharm.



Pytest - Test simple

```
# test_SomeFunction.py
def test_SomeFunction():
    assert 1 == 1
```

- “test” al inicio de la función
- Usan assert de Python
- Se pueden agrupar dentro de un módulo o clase.

Pytest - Test discovery

Pytest reconoce automáticamente los siguientes casos:

- funciones con “test” al inicio del nombre
- Clases con “Test” al inicio del nombre y sin constructor
- Módulos que comiencen o terminen con “test”.

Pytest - XUnit style setup/teardown

Funciones que se ejecutan antes (setup) y después (teardown) de los tests, tanto para funciones, clases y módulos:

- `setup_function()`
- `teardown_function()`
- `setup_class()`
- `teardown_class()`
- etc.

Pytest - Fixture setup

```
@pytest.fixture():  
def math():  
    return Math()  
  
def test_Add(math):  
    assert math.add(1,1) == 2
```

- Permite reutilizar funciones de pre y post procesamiento para distintos tests.
- `pytest.fixture` decorator
- Cada test puede indicar qué fixture utilizar.
- `autouse=True` ejecuta el fixture automáticamente para todos los tests.

Pytest - Fixture teardown

```
@pytest.fixture():  
def setup():  
    print("Setup!")  
    yield  
    print("Teardown!")
```

- Todo código posterior a yield se ejecuta cuando el fixture sale del scope. Yield actúa como reemplazo de return.

Pytest - Fixture teardown

```
@pytest.fixture():  
def setup(request):  
    print("Setup!")  
    def teardown():  
        print("Teardown!")  
    request.addfinalizer(teardown)
```

- Utiliza el request-context.
- A diferencia de yield permite utilizar varias funciones teardown.

Pytest - Fixture parámetros

```
@pytest.fixture(params=[1,
2]):
def setupData(request):
    return request.params

def test_1(setupData):
    print(setupData)
```

- Fixtures pueden entregar datos
- Se puede usar params para indicar los datos a devolver
- Los tests corren una vez por juego de parámetros.

Pytest - Fixture scope

- Función: se ejecuta para cada función
- Clase: se ejecuta para cada clase Test
- Módulo: se ejecuta una vez en el módulo
- Sesión: se ejecuta una vez cuando inicia pytest

Pytest - Assert

- ▷ Comparación standard:
 - `>`, `<`, `==`, `!=`, etc.
- ▷ Valores con punto flotante
 - `approx(3.14159)`
- ▷ `with raises` para verificar que un `assert` dispare una excepción esperada.

Pytest - CLI

Por defecto pytest corre todos los test que encuentra bajo los criterios que vimos anteriormente. Se puede indicar qué tests correr con los siguientes argumentos:

- ▷ `moduleName`
- ▷ `DirectoryName/`
- ▷ `-k "expression"` (busca la expresión en el directorio)
- ▷ `-m "expression"` (similar al anterior pero usando el decorator `pytest.mark`)

Pytest - CLI

Argumentos adicionales

- ▷ -v: reporte “verbose”
- ▷ -q: quiet mode
- ▷ -s: no muestra salidas por consola
- ▷ --ignore: ignorar un directorio para no correr tests.
- ▷ --maxfail: indicar a pytest que frene a partir de un número de fallas

Pytest - Extras

- ▷ Dummies
- ▷ Fakes
- ▷ Stubs
- ▷ Spies
- ▷ Mocks
- ▷ monkeypatch

Ejemplo: Cajero Supermercado

- ▷ Clase Checkout que mantiene la lista de items a cobrar.
- ▷ Funciones:
 - Setear el precio individual de un ítem
 - Agregar items individuales.
 - Calcular el costo total para todos los items agregados
 - Agregar y aplicar descuentos cuando N ítems de un tipo se agregan.

Ejemplo: Cajero Supermercado

- ▷ Test cases:
 - Se puede crear una instancia de la clase
 - Se puede agregar el precio de un ítem
 - Se puede agregar un ítem
 - Se puede calcular el total
 - Se pueden agregar varios ítems y calcular el total
 - Se pueden agregar reglas de descuento
 - Se puede aplicar el descuento al total
 - Se dispara una excepción si se agrega un ítem sin precio.

Documentación





Code is more often read than written.

— Guido van Rossum

Comentarios vs. Documentación

- ▷ Los comentarios generalmente son para los desarrolladores
- ▷ La documentación, en su mayoría, es para los usuarios.

Comentarios

▷ Planificación y revisión

Python

```
# First step  
# Second step  
# Third step
```

Comentarios

▷ Descripción del código

Python

```
# Attempt a connection based on previous settings. If unsuccessful,  
# prompt user for new settings.
```

Comentarios

- ▷ Descripción del algoritmo

Python

```
# Using quick sort for performance gains
```

Comentarios

- ▷ Tagging (TODO; BUG, FIXME, etc.)

Python

```
# TODO: Add condition for when val is None
```

Docstrings

Python

```
def say_hello(name):  
    """A simple function that says hello... Richie style"""  
    print(f"Hello {name}, is it me you're looking for?")
```

Python

>>>

```
>>> help(say_hello)  
Help on function say_hello in module __main__:  
  
say_hello(name)  
    A simple function that says hello... Richie style
```


Docstrings

- ▷ Las convenciones están en [PEP 257](#).
- ▷ Los docstring multilínea generalmente tienen la siguiente estructura:
 - un resumen de una línea
 - una línea vacía
 - el resto del docstring
 - otra línea vacía previa al código

Docstrings

Se pueden distinguir los docstrings en diferentes niveles de aplicación:

- Clases
- Paquetes y módulos
- Scripts

Docstrings - Clases

General

- Breve resumen del propósito y funcionamiento
- Todo método público con una breve descripción
- Cualquier propiedad
- Todo lo relacionado a la interfaz, para el caso de subclases

Métodos

- Breve descripción del método y propósito
- Todos los argumentos (opcionales y requeridos), incluido los keywords.
- Etiquetar los argumentos opcionales y con valores por defecto
- Cualquier efecto secundario
- Indicar las excepciones
- Indicar restricciones para ejecutar el método

Docstrings - Clases

General

```
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    -----
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -----
    says(sound=None)
        Prints the animals name and what sound it makes
    """
```

Métodos

```
def says(self, sound=None):
    """Prints what the animals name is and what sound it makes.

    If the argument `sound` isn't passed in, the default Animal
    sound is used.

    Parameters
    -----
    sound : str, optional
        The sound the animal makes (default is None)

    Raises
    -----
    NotImplementedError
        If no sound is set for the animal or passed in as a
        parameter.
    """

    if self.sound is None and sound is None:
        raise NotImplementedError("Silent Animals are not supported!")

    out_sound = self.sound if sound is None else sound
    print(self.says_str.format(name=self.name, sound=out_sound))
```

Docstrings - Tipos

Formatting Type	Description	Supported by Sphinx	Formal Specification
Google docstrings	Google's recommended form of documentation	Yes	No
reStructured Text	Official Python documentation standard; Not beginner friendly but feature rich	Yes	Yes
NumPy/SciPy docstrings	NumPy's combination of reStructured and Google Docstrings	Yes	Yes
Epytext	A Python adaptation of Epydoc; Great for Java developers	Not officially	Yes

reStructured Text Example

Python

```
"""Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
"""
```

Python

```
"""Gets and prints the spreadsheet's header columns

Parameters
-----
file_loc : str
    The file location of the spreadsheet
print_cols : bool, optional
    A flag used to print the columns to the console (default is False)

Returns
-----
list
    a list of strings representing the header columns
"""
```

Documentación - Open Source

- ▷ Readme
- ▷ How to Contribute
- ▷ Code of Conduct
- ▷ License
- ▷ docs

Documentación - Extras

Es una [buena práctica](#) agregar una carpeta 'docs' con documentación adicional, independientemente del tipo de proyecto.

- ▷ Tutorials
- ▷ How-To guides
- ▷ References
- ▷ Explanations

Sphinx

Primeros pasos:

- `$ pip install sphinx`
- `$ pip install sphinx-rtd-theme`
- `$ cd /path/to/project`
- `$ mkdir docs`

Sphinx

Creamos las configuraciones:

- `$ cd docs`
- `$ sphinx-quickstart`

- ▷ `config.py`
- ▷ `index.rst`
- ▷ `Makefile`

Sphinx

Build

- `$ make clean`
- `$ make html`
- `$ python -m http.server`

En el localhost se puede revisar la documentación generada

Sphinx

Build

- `$ make clean`
- `$ make build`
- `$ python -m http.server`

En el localhost se puede revisar la documentación generada

Sphinx - Source Format

reStructuredText	Markdown	Notebooks
Sin cambios en conf.py	usar recommonmark en conf.py	usar nbsphinx en conf.py
Sin recomendaciones	Usar MyST	Sin recomendaciones
Se pueden usar todos juntos. Se puede usar pandoc para convertir de md a rst.*		

You can mix and match source formats.

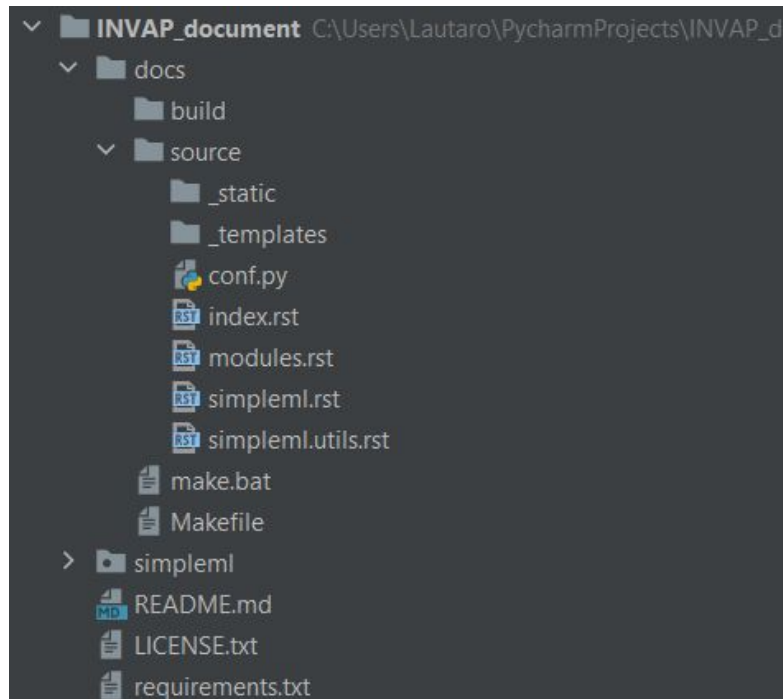
```
.. toctree::  
    :maxdepth: 2  
    :caption: Contents:
```

```
sample_doc.rst  
markdown_doc.md  
notebook.ipynb
```

Sphinx - Documentación Docstrings

Dentro del directorio de docs, ejecutamos los siguiente:

```
$ sphinx-apidoc -o  
./source ../{name}
```



Sphinx - autodocs

Submodules

simpleml.kmeans module

```
.. automodule:: simpleml.kmeans
   :members:
   :undoc-members:
   :show-inheritance:
```

simpleml.linear_regression module

```
.. automodule:: simpleml.linear_regression
   :members:
   :undoc-members:
```

simpleml.utils package

Submodules

simpleml.utils.metrics module

```
.. automodule:: simpleml.utils.metrics
   :members:
   :undoc-members:
   :show-inheritance:
```

Module contents

```
.. automodule:: simpleml.utils
```

[autodoc](#)
[python directives](#)

Sphinx - autodoc

- ▷ Se pueden modificar los rst. generados automáticamente para mayor claridad, agregar ecuaciones, imágenes, etc.
- ▷ Se pueden crear nuevos .rst para agregar a la documentación:
 - Introducción
 - Ejemplos
 - Tutoriales
 - etc.

Sphinx

Build final

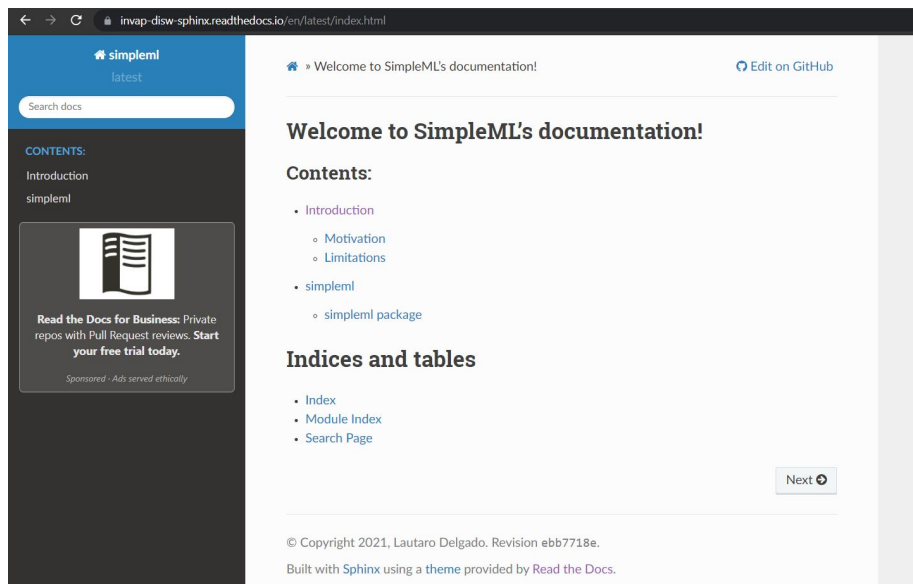
- `$ make clean`
- `$ make html`

Sphinx Publicación en RTD

- ▷ Publicar el repositorio en github
- ▷ Con github:
 - Log in en Read The Docs
 - Vincular el repositorio
- ▷ [Sin github](#)
- ▷ Agregar requirements.txt en settings

Sphinx Publicación en RTD

▷ Documentación generada



Publicación de Paquetes



COPY&PASTE
IS NOT
HOW YOU SHOULD
SHARE CODE



Publishing PyPi

Introducción

```
def say_hello(name=None):  
    if name is None:  
        return "Hello, World! "  
    else:  
        return f"Hello, {name} !"
```

Setuptools

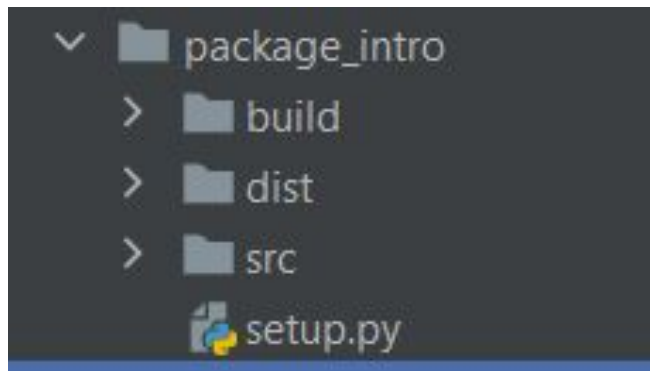
Utilizamos [setuptools](#) para generar un archivo `setup.py` con la metadata de nuestro paquete.

```
setup(  
    name='helloworld',  
    version='0.0.1',  
    description='Say hello!',  
    py_modules=["helloworld"],  
    package_dir={'': 'src'},  
)
```

Build

```
$python setup.py bdist wheel
```

–python wheel



Instalar localmente

```
$pip install -e .
```

```
(base) C:\Users\Lautaro\PycharmProjects\INVAP_Python\package_intro>pip install -e .  
Obtaining file:///C:/Users/Lautaro/PycharmProjects/INVAP\_Python/package\_intro  
Installing collected packages: helloworld  
  Running setup.py develop for helloworld  
Successfully installed helloworld
```

Test local

```
$python
```

```
>>> from helloworld import say_hello
```

```
>>> say_hello()
```

.gitignore



Create useful .gitignore files for your project

Create

[Source Code](#)

| [Command Line Docs](#)

LICENSE

choosealicense.com

Choose an open source license

An open source license protects contributors and users. Businesses and savvy developers won't touch a project without this protection.

{ Which of the following best describes your situation? }



I need to work in a community.

Use the [license preferred by the community](#) you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to [add a license](#).



I want it simple and permissive.

The [MIT License](#) is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

[Babel](#), [.NET Core](#), and [Rails](#) use the MIT License.



I care about sharing improvements.

The [GNU GPLv3](#) also lets people do almost anything they want with your project, *except* distributing closed source versions.

[Ansible](#), [Bash](#), and [GIMP](#) use the GNU GPLv3.

LICENSE.txt :

- MIT License
- GNU GPLv3
- etc.

Setup classifiers

Lista de classifiers

```
classifiers=[  
    "Programming Language :: Python :: 3",  
    "License :: OSI Approved :: MIT License",  
    "Operating System :: OS Independent",  
]
```

Agregar README a setup.py

```
with open("Readme.md", "r") as fh:
    long_description = fh.read()

setup(
    ...
    long_description = long_description,
    long_description_content_type = "text/markdown"
)
```

Dependencias install

```
install_requires = [  
    "blessings ~= 1.7",  
],
```

Dependencias development

```
extras_require = {  
    "dev": [  
        "pytest >= 3.7",  
        "check-manifest",  
        "twine",  
    ],  
}
```

```
$ pip install -e .[dev]
```


Dependencias Install vs. Extras

Install:

- dependencias para producción (NumPy, Flask, etc.)
- las versiones deberían todo lo laxas posible (>3.0, etc.)

Extras:

- requerimientos opcionales (Pytest, Mock, Coverage, etc.)
- versiones explícitas

Requirements.txt

- Para entornos controlables
- Se indican versiones fijas
- `pip freeze > requirements.txt`

Testing

- Incluimos tests en pytest dentro del paquete

```
(base) C:\Users\Lautaro\PycharmProjects\INVAP_Python\package_intro>pytest
===== test session starts =====
platform win32 -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\Lautaro\PycharmProjects\INVAP_Python\package_intro
plugins: asyncio-0.14.0, cov-2.7.1, docker-compose-3.0.0, dotenv-0.5.2, html-1.22.0, metadata-1.11.0, mock-3.3.1, ordering-0.6, requests-mock-1.8.0
collected 2 items

test_helloworld.py .. [100%]

===== 2 passed in 0.06s =====
```

Source distribution

```
$ python setup.py sdist
```

- Acceso al código fuente
- Diferencia con binary distribution

Se debe indicar:

- web del proyecto, autor y su email
- por defecto no incluye los test, licencia, etc.
- → crear un MANIFEST.in

MANIFEST.in

```
$ pip install check-manifest
```

```
$ check-manifest --create
```

```
$ git add MANIFEST.in
```

Build final

```
$ python setup.py bdist_wheel sdist  
$ ls dist/
```

Publicación

- Crear un usuario en PyPi
- Configurar un token
- `twine upload dist/*`

Se puede usar como [alternativa](#) TestPyPI cuando se están realizando pruebas.

Publicación

New releases

Hot off the press: the newest project releases



helloworld-disw 0.0.1

Say hello!



vsrealesrgan 1.0.0

RealESRGAN function for VapourSynth



pollination-point-in-time-view 0.2.4

Point-in-time view-based recipe for Pollination.




fastip 0.0.2


SDK about fastip



melib 1.1.0

Library functions to help with machine element design projects.



 **lautaroDC**

helloworld-disw 0.0.1

`pip install helloworld-disw`

Released: half a minute ago

Say hello!

Manage project

Navigation

- Project description
- Release history
- Download files

Project description

Hello World

This is an example project demonstrating how to publish a python module to PyPi.

Installation

Run the following to install:

Extras

- ▷ Tox: testear diferentes versiones de python
- ▷ Travis, etc: CI workflows
- ▷ Badges:
 - Code coverage: codecov.io
 - Code quality: Code Climate
- ▷ [bumpversion](#)

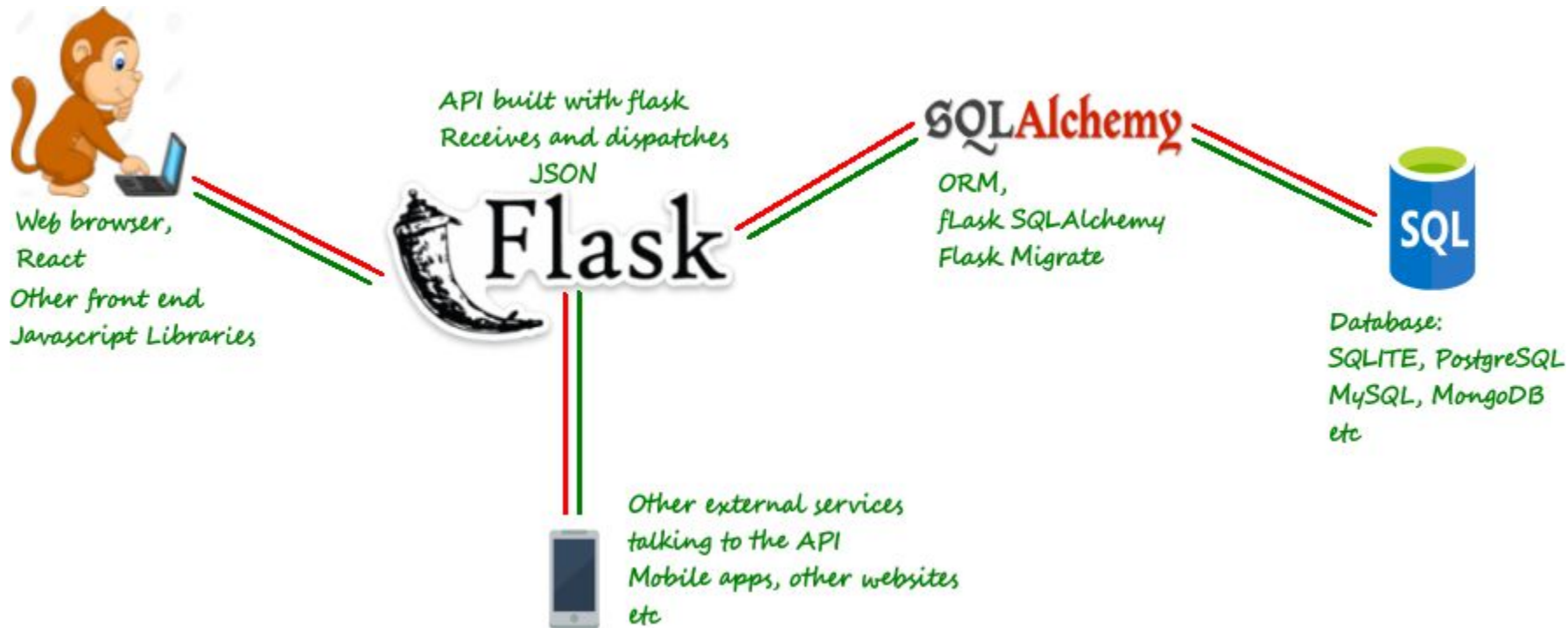
Alternativas

- ▷ Mover [metadata](#) de setup.py a setup.cfg
- ▷ Metadata en pyproject.toml:
 - Poetry
 - Flit
 - Hatch
 - etc

Desarrollo de APIs

Flask





Fuente: [What is an API?](#)

Design - Projects

POST /design/projects Create a new item

GET /design/projects/{id} Find an item by ID

PUT /design/projects/{id} Update an item by ID

DELETE /design/projects/{id} Delete an item by ID

POST /design/projects/all Lists tests by ids

GET /design/projects/by-workspace/{workspaceId}/{type} List projects by workspace ID and type

APIs en Python

Existen varias opciones para desarrollar APIs en Python, con diferente nivel de potencia:

- ▷ Flask
- ▷ Django
- ▷ Fast API
- ▷ etc.

Ejemplo: Youtube API Mock

- ▷ Creamos un nuevo entorno virtual
- ▷ Instalamos los paquetes necesarios:

```
aniso8601==8.0.0
click==7.1.2
Flask==1.1.2
Flask-RESTful==0.3.8
Flask-SQLAlchemy==2.4.3
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
pytz==2020.1
six==1.15.0
SQLAlchemy==1.3.18
Werkzeug==1.0.1
```

Ejemplo mínimo

▷ main.py

```
from flask import Flask
from flask_restful import Api, Resource

app = Flask(__name__)
api = Api(app)

if __name__ == '__main__':
    app.run(debug=True)
```


Resource

Clases que heredan de Resource para indicar el comportamiento para un determinado request: get, post, delete, etc.

```
class HelloWorld(Resource):  
    def get(self):  
        return {"Hello World"}  
  
api.add_resource(HelloWorld, "/helloworld")
```

Testing requests

Usamos la librería request de python:

```
import requests

BASE = "http://127.0.0.1:5000/"

response = requests.get(BASE + "helloworld")
print(response.json())
```

Pasando argumentos

- Ejemplos de argumentos

```
class HelloWorld(Resource):  
    def get(self, name):  
        return {"data": name}  
  
api.add_resource(HelloWorld, "/helloworld/<string:name>")
```

Parsear argumentos

```
video_put_args = reqparse.RequestParser()
video_put_args.add_argument("name", type=str, help="Name of the video is required", required=True)
video_put_args.add_argument("views", type=int, help="Views of the video", required=True)
video_put_args.add_argument("likes", type=int, help="Likes on the video", required=True)

videos = {}

class Video(Resource):
    def get(self, video_id):
        return videos[video_id]

    def put(self, video_id):
        args = video_put_args.parse_args()
        return {video_id: args}

api.add_resource(Video, "/video/<int:video_id>")

if __name__ == "__main__":
    app.run(debug=True)
```

flask_restful viene con un parser de argumentos incluido “reqparse”, para tratar los datos entrantes.

Código de status

```
videos = {}

class Video(Resource):
    def get(self, video_id):
        return videos[video_id]

    def put(self, video_id):
        args = video_put_args.parse_args()
        videos[video_id] = args
        return videos[video_id], 201

api.add_resource(Video, "/video/<int:video_id>")
```

Existen varios
códigos de estado,
ver [referencias](#).

Validando requests

```
videos = {}

def abort_if_video_id_doesnt_exist(video_id):
    if video_id not in videos:
        abort(404, message="Could not find video...")

class Video(Resource):
    def get(self, video_id):
        abort_if_video_id_doesnt_exist(video_id)
        return videos[video_id]

    def put(self, video_id):
        args = video_put_args.parse_args()
        videos[video_id] = args
        return videos[video_id], 201
```

Usando bases de datos

En éste caso vamos a usar una implementación simple con SQLite. En todos los casos, se suele usar con Flask SQLAlchemy:

- `$ pip install flask-sqlalchemy`

En caso de usar otra base de datos, se debe instalar librerías extras, ej:

- PostgreSQL → `psycopg2`

Configurar la base de datos

```
app = Flask(__name__)  
api = Api(app)  
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'  
db = SQLAlchemy(app)
```

```
from flask_migrate import Migrate  
  
app = Flask(__name__)  
app.config['SQLALCHEMY_DATABASE_URI'] = "postgresql://postgres:postgres@localhost:5432/cars_api"  
db = SQLAlchemy(app)  
migrate = Migrate(app, db)
```


Creando modelos

```
class VideoModel(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    views = db.Column(db.Integer, nullable=False)
    likes = db.Column(db.Integer, nullable=False)

    def __repr__(self):
        return f"Video(name = {name}, views = {views}, likes = {likes})"

db.create_all()
```

Querying y Serialización

```
resource_fields = {
    'id': fields.Integer,
    'name': fields.String,
    'views': fields.Integer,
    'likes': fields.Integer
}

class Video(Resource):
    @marshal_with(resource_fields)
    def get(self, video_id):
        result = VideoModel.query.filter_by(id=video_id).first()
        return result
```

El decorator `marshal_with` nos permite serializar el objeto respuesta de la query, usando un diccionario (`resource_fields` en éste caso). El atributo `fields`, es propio de `flask_restful` e indica el tipo de dato.

Querying y Serialización

```
@marshal_with(resource_fields)
def put(self, video_id):
    args = video_put_args.parse_args()
    result = VideoModel.query.filter_by(id=video_id).first()
    if result:
        abort(409, message="Video id taken...")

    video = VideoModel(id=video_id, name=args['name'], views=args['views'], likes=args['likes'])
    db.session.add(video)
    db.session.commit()
    return video, 201
```

Bibliografía



Documentación (docstrings, estructura, etc.)

Documentación ReadTheDocs

Documentación Sphinx

Unit Testing and Test Driven Development in Python

Patrones de diseño: Teoría

Patrones de diseño: Práctica

Publicación de paquetes

Flask API