

Python Intermedio



El Zen de Python



¿Qué es?



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, ¡tenemos más de esos!

¿Qué es la
documentación?



Y por qué es tan importante

¿Qué es un entorno virtual?



Controla tus módulos

Mi computadora

Python



Mi computadora

Python

Proyecto 1

Módulo 1

Módulo 2

Módulo 3

Python

Proyecto 2

Módulo 1

Módulo 2

Módulo 3

Python

Proyecto 3

Módulo 1

Módulo 2

Módulo 3

Python

Proyecto 4

Módulo 1

Módulo 2

Módulo 3

Mi computadora

Python

Proyecto 1

Módulo 1

Módulo 2 v2

Módulo 3

Python

Proyecto 2

Módulo 1

Módulo 2 v2

Módulo 3

Python

Proyecto 3

Módulo 1

Módulo 2 v2

Módulo 3

Python

Proyecto 4

Módulo 1

Módulo 2

Módulo 3

Python

Proyecto 1

Módulo 1

Módulo 2 v2

Módulo 3

Instalación de dependencias con PIP

Package Installer for Python

Listas y diccionarios anidados



List comprehensions

Genera listas sin ciclos

```
[element for element in iterable if condition]
```



```
[element for element in iterable if condition]
```

Representa
a cada uno
de los
elementos
a poner en
la nueva
lista

Ciclo a partir del cual se extraerán
elementos de otra lista o cualquier
iterable

Condición
opcional para
filtrar los
elementos del
ciclo

```
[i**2 for i in range(1, 101) if i % 3 != 0]
```



Representa
a cada uno
de los
elementos
a poner en
la nueva
lista

Ciclo a partir del cual se extraerán
elementos de otra lista o cualquier
iterable

Condición
opcional para
filtrar los
elementos del
ciclo

Dictionary comprehensions

Genera diccionarios sin ciclos

```
{key:value for value in iterable if condition}
```

```
{key:value for value in iterable if condition}
```

Representa a
cada una de
las llaves y
valores a
poner en el
nuevo
diccionario

Ciclo a partir del cual se
extraerán elementos de
cualquier iterable

Condición
opcional para
filtrar los
elementos del
ciclo

```
{i: i**3 for i in range(1, 101) if i % 3 != 0}
```

Representa
a cada una
de las llaves
y valores a
poner en el
nuevo
diccionario

Ciclo a partir del cual se extraerán
elementos de cualquier iterable

Condición
opcional para
filtrar los
elementos del
ciclo

Funciones anónimas

Y qué significa “lambda”



```
lambda argumentos: expresión
```




```
palindrome = lambda string: string == string[::-1]  
print(palindrome('ana'))
```



True



identificador

argumento

expresión

```
palindrome = lambda string: string == string[::-1]
```

```
print(palindrome('ana'))
```



True



retorna un objeto de tipo función

```
palindrome = lambda string: string == string[::-1]
```

```
print(palindrome('ana'))
```



True



```
# palindrome = lambda string: string == string[::-1]

def palindrome(string):
    return string == string[::-1]

print(palindrome('ana'))
```



True

High order functions

Filter, map y reduce

Función de orden superior

Es una función que recibe como parámetro a otra función.



```
def saludo(func):  
    func()  
  
def hola():  
    print("Hola!!!")  
  
def adios():  
    print("Adios!!!")  
  
saludo(hola)  
saludo(adios)
```



```
Hola!!!  
Adios!!!
```

filter

[1, 4, 5, 6, 9, 13, 19, 21]



[1, 5, 9, 13, 19, 21]



```
my_list = [1, 4, 5, 6, 9, 13, 19, 21]  
odd = [i for i in my_list if i % 2 != 0]  
print(odd)
```



```
[1, 5, 9, 13, 19, 21]
```



```
# Uso con filter
```

```
my_list = [1,4,5,6,9,13,19,21]
```

```
odd = list(filter(lambda x: x%2 != 0, my_list))
```

```
print(odd)
```



```
[1, 5, 9, 13, 19, 21]
```

m a p

[1, 2, 3, 4, 5]



[1, 4, 9, 16, 25]



```
my_list = [1, 2, 3, 4, 5]  
squares = [i**2 for i in my_list]  
print(squares)
```



```
[1, 4, 9, 16, 25]
```



```
# Uso con map  
  
my_list = [1, 2, 3, 4, 5]  
  
squares = list(map(lambda x: x**2, my_list))  
  
print(squares)
```



```
[1, 4, 9, 16, 25]
```

reduce

$[2, 2, 2, 2, 2]$



32



```
my_list = [2, 2, 2, 2, 2]

all_multiplied = 1

for i in my_list:
    all_multiplied = all_multiplied * i

print(all_multiplied)
```



32



```
# Use con reduce
```

```
from functools import reduce
```

```
my_list = [2, 2, 2, 2, 2]
```

```
all_multiplied = reduce(lambda a, b: a * b, my_list)
```

```
print(all_multiplied)
```



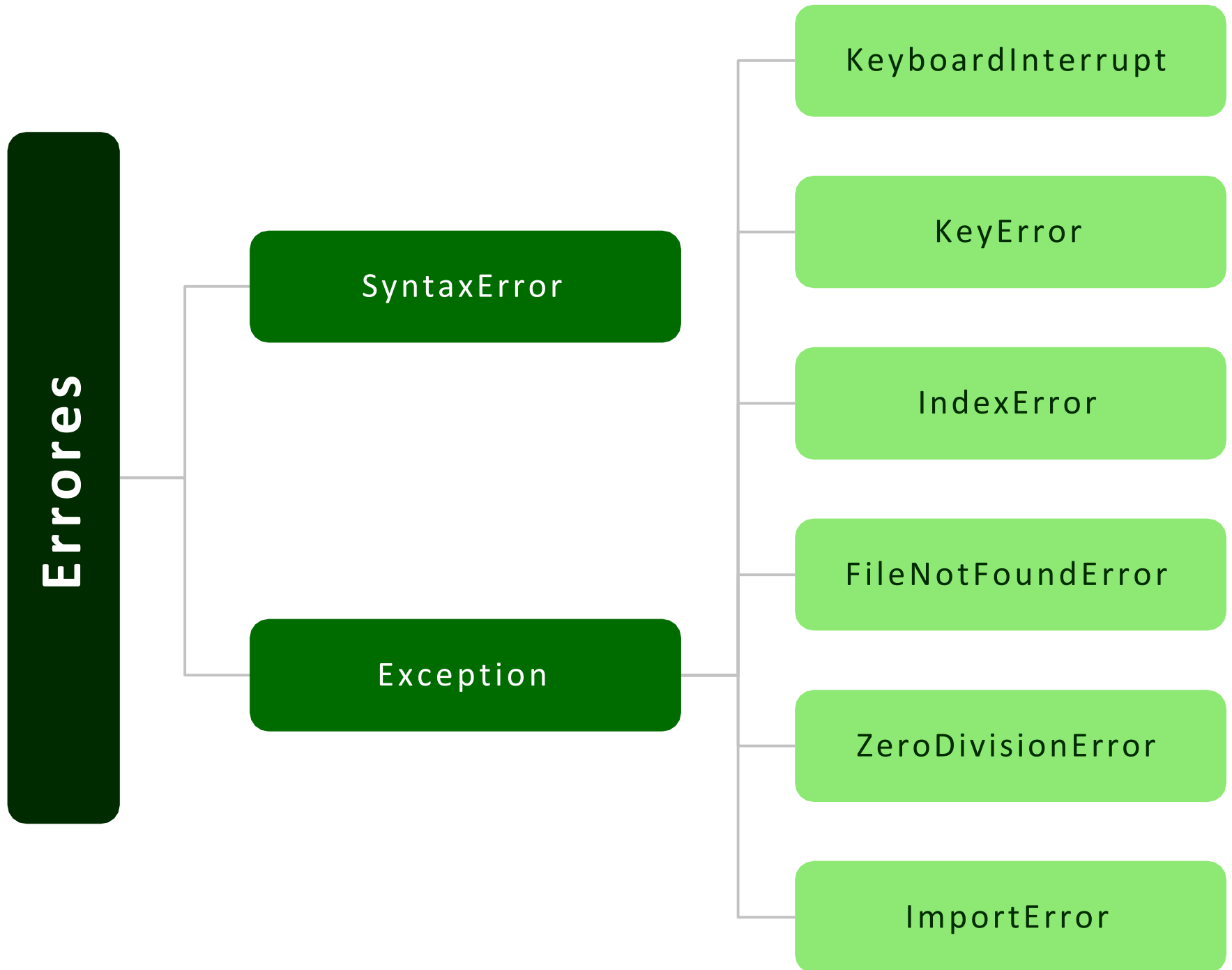
```
32
```

Practica

¡Pongamos en práctica lo aprendido!

Los errores en el código

Y cómo manejarlos



```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```


Debugging

O también, depuración


Manejo de excepciones

raise, try, except y finally

try, except



```
def palindrome(string):  
    return string == string[::-1]  
  
print(palindrome(1))
```



```
Traceback (most recent call last):  
  File "main.py", line 4, in <module>  
    print(palindrome(1))  
  File "main.py", line 2, in palindrome  
    return string == string[::-1]  
TypeError: 'int' object is not subscriptable
```



```
def palindrome(string):  
    return string == string[::-1]  
  
try:  
    print(palindrome(1))  
except TypeError:  
    print("Solo se pueden ingresar strings")
```



Solo se pueden ingresar strings


raise



```
def palindrome(string):  
    return string == string[::-1]  
  
try:  
    print(palindrome(""))  
except TypeError:  
    print("Solo se pueden ingresar strings")
```



True



```
def palindrome(string):  
    try:  
        if len(string) == 0:  
            raise ValueError("No se puede ingresar una cadena vacía")  
        return string == string[::-1]  
    except ValueError as ve:  
        print(ve)  
        return False  
  
try:  
    print(palindrome(""))  
except TypeError:  
    print("Solo se pueden ingresar strings")
```



```
No se puede ingresar una cadena vacía  
False
```

finally



```
try:
    f = open("archivo.txt")
    # hacer cualquier cosa con nuestro archivo
finally:
    f.close()
```

Assert Statements

Afirmaciones en Python

código



```
graph TD; A[código] --> B[Aserción]; B --> C[error]; B --> D[código];
```

Aserción

error

código



```
assert condición, mensaje de error
```



```
def palindrome(string):  
    return string == string[::-1]  
  
print(palindrome(""))
```



True



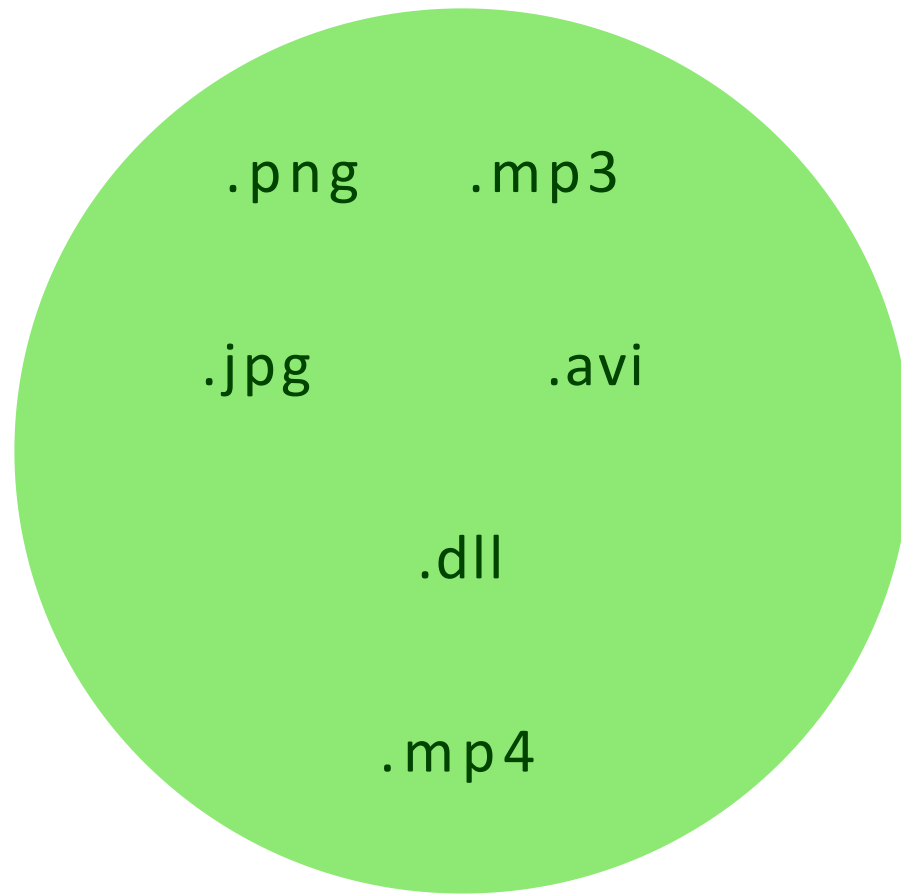
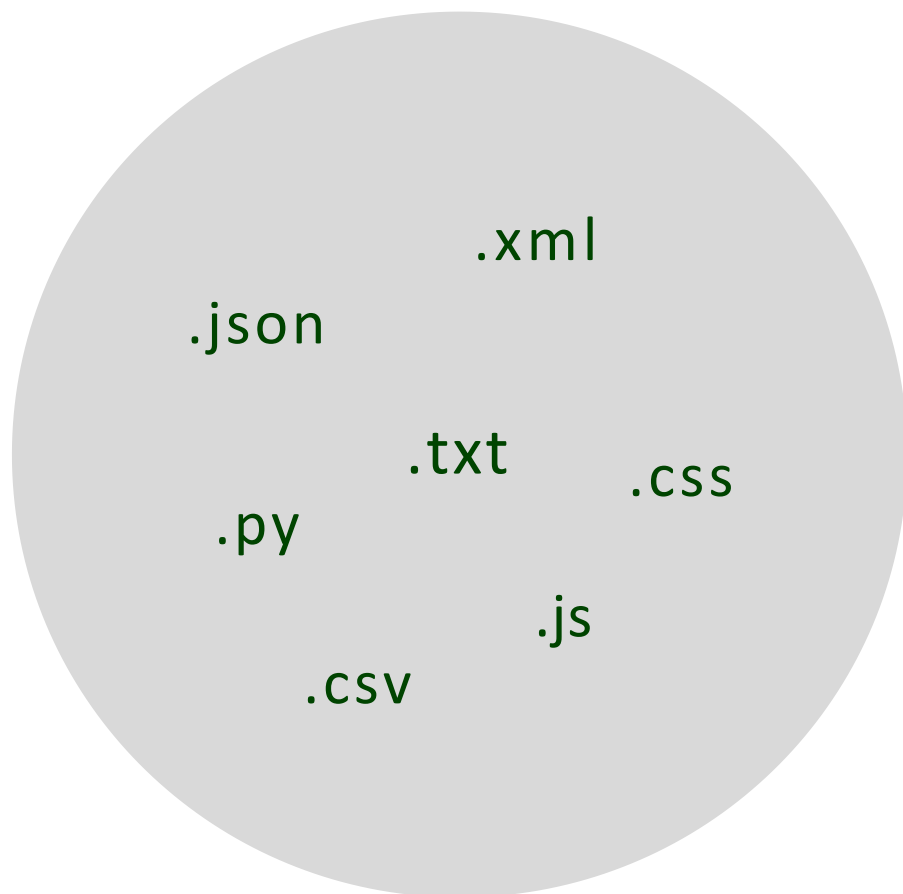
```
def palindrome(string):  
    assert len(string) > 0, "No se puede ingresar una cadena vacía"  
    return string == string[::-1]  
  
print(palindrome(""))
```

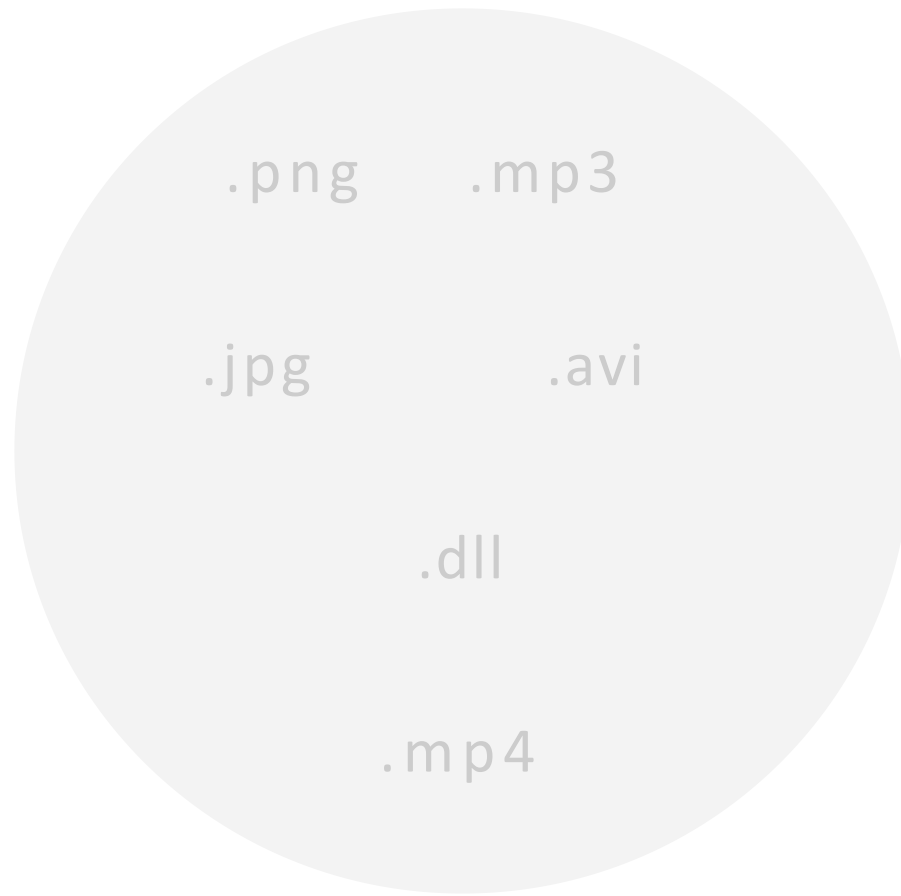
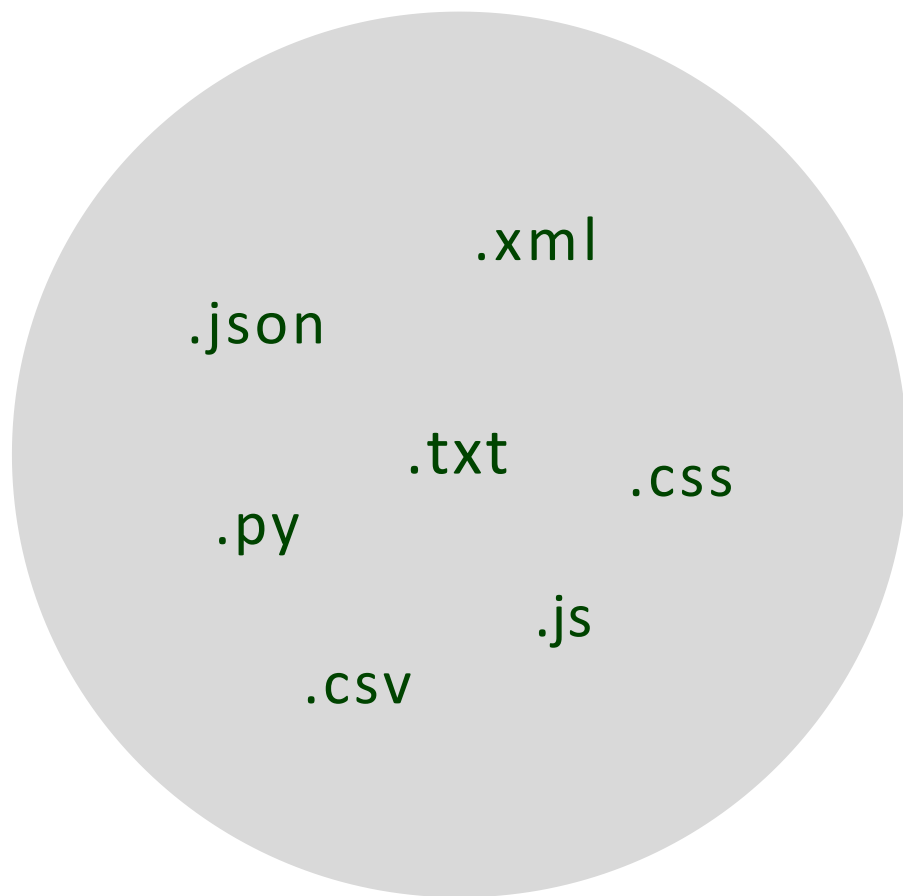


```
AssertionError: No se puede ingresar una cadena vacía
```

¿Cómo trabajar con archivos?

Lectura y escritura





Modos de apertura

- **R** -> Lectura
- **W** -> Escritura (sobrescribir)
- **A** -> Escritura (agregar al final)



```
with open("./ruta/del/archivo.txt", "r") as f:
```