# CS50 PYTHON

## match

> Similar to `if`, `elif`, and `else` statements, `match` statements can be used to conditionally run code that matches certain values.

Consider the following program:

```
name = input("What's your name? ")

if name == "Harry":
    print("Gryffindor")
elif name == "Hermione":
    print("Gryffindor")
elif name == "Ron":
    print("Gryffindor")
elif name == "Draco":
    print("Slytherin")
else:
    print("Who?")
```

> Notice the first three conditional statements print the same response.

We can improve this code slightly with the use of the `or` keyword:

```
name = input("What's your name? ")

if name == "Harry" or name == "Hermione" or name == "Ron":
    print("Gryffindor")
elif name == "Draco":
    print("Slytherin")
else:
    print("Who?")
```

> Notice the number of `elif` statements has decreased, improving the readability of our code.

Alternatively, we can use `match` statements to map names to houses. Consider the following code:

```
name = input("What's your name? ")

match name:
```

```
    case "Harry":
        print("Gryffindor")
    case "Hermione":
        print("Gryffindor")
    case "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

Notice the use of the _ symbol in the last case. This will match with any input, resulting in similar behavior as an else statement.
A match statement compares the value following the match keyword with each of the values following the case keywords. In the event a match is found, the respective indented code section is executed and the program stops the matching.

We can improve the code:

```
name = input("What's your name? ")

match name:
    case "Harry" | "Hermione" | "Ron":
        print("Gryffindor")
    case "Draco":
        print("Slytherin")
    case _:
        print("Who?")
```

Notice, the use of the single vertical bar |. Much like the or keyword, this allows us to check for multiple values in the same case statement.

# Dictionaries

dicts or dictionaries is a data structure that allows you to associate keys with values.
Where a list is a list of multiple values, a dict associates a key with a value. Considering the houses of Hogwarts, we might assign specific students to specific houses.

| Hermione | Harry | Ron | Draco |
|---|---|---|---|
| Gryffindor | Gryffindor | Gryffindor | Slytherin |

We could use lists alone to accomplish this:

```
students = ["Hermoine", "Harry", "Ron", "Draco"]
houses = ["Gryffindor", "Gryffindor", "Griffindor", "Slytherin"]
```

> Notice that we could promise that we will always keep these lists in order. The individual at the first position of students is associated with the house at the first position of the houses list, and so on. However, this can become quite cumbersome as our lists grow!

We can better our code using a dict as follows:

```
students = {
    "Hermoine": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin",
}
print(students["Hermoine"])
print(students["Harry"])
print(students["Ron"])
print(students["Draco"])
```

> Notice how we use {} curly braces to create a dictionary. Where lists use numbers to iterate through the list, dicts allow us to use words.

Run your code and make sure your output is as follows:

```
$ python hogwarts.py
Gryffindor
Gryffindor
Gryffindor
Slytherin
```

We can improve our code as follows:

```
students = {
    "Hermoine": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin",
}
for student in students:
    print(student)
```

> Notice how executing this code, the for loop will only iterate through all the keys, resulting in a list of the names of the students. How could we print out both values and keys?

Modify your code as follows:

```
students = {
```

```
    "Hermoine": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin",
}
for student in students:
    print(student, students[student])
```

> Notice how `students[student]` will go to each student's key and find the value of the their house. Execute your code and you'll notice how the output is a bit messy.

We can clean up the print function by improving our code as follows:

```
students = {
    "Hermoine": "Gryffindor",
    "Harry": "Gryffindor",
    "Ron": "Gryffindor",
    "Draco": "Slytherin",
}
for student in students:
    print(student, students[student], sep=", ")
```

> Notice how this creates a clean separation of a , between each item printed.

If you execute `python hogwarts.py`, you should see the following:

```
$ python hogwarts.py
Hermoine, Gryffindor
Harry, Gryffindor
Ron, Gryffindor
Draco, Slytherin
```

What if we have more information about our students? How could we associate more data with each of the students?

| | name | house | patronus |
|---|---|---|---|
| 0 | Hermione | Gryffindor | Otter |
| 1 | Harry | Gryffindor | Stag |
| 2 | Ron | Gryffindor | Jack Russell terrier |
| 3 | Draco | Slytherin | |

You can imagine wanting to have lots of data associated multiple things with one key. Enhance your code as follows:

```
students = [
    {"name": "Hermoine", "house": "Gryffindor", "patronus": "Otter"},
    {"name": "Harry", "house": "Gryffindor", "patronus": "Stag"},
    {"name": "Ron", "house": "Gryffindor", "patronus": "Jack Russell terrier"},
    {"name": "Draco", "house": "Slytherin", "patronus": None},
]
```

Notice how this code creates a list of dicts. The list called students has four dicts within it: One for each student. Also, notice that Python has a special None designation where there is no value associated with a key.

Now, you have access to a whole host of interesting data about these students. Now, further modify you code as follows:

```
students = [
    {"name": "Hermoine", "house": "Gryffindor", "patronus": "Otter"},
    {"name": "Harry", "house": "Gryffindor", "patronus": "Stag"},
    {"name": "Ron", "house": "Gryffindor", "patronus": "Jack Russell terrier"},
    {"name": "Draco", "house": "Slytherin", "patronus": None},
]

for student in students:
    print(student["name"], student["house"], student["patronus"], sep=", ")
```

Notice how the for loop will iterate through each of the dicts inside the list called students.
You can learn more in Python's Documentation of dicts.

# LIBRARIES

https://cs50.harvard.edu/python/2022/notes/4/#libraries

Libraries seen:

- Random
- Stratistics
- packages
- sys

## Sys

sys: system
sys.argv -> sys. argument variable
ej:
**import sys**
**print(“hello, my name is”, sys.argv[1])**

Terminal: $ python name.py Matias
devuelve: hello, my name is Matias

en el [0] devuelve el nombre del archivo (name.py)

sys.exit(“chau”) -> termina el programa y devuelve un “chau”

No se pueden usar muchos elses, en vez de else usar elif y en ultimo caso else.

# Random

randint -> da un num random

ej: number = random.randint(1, 10)
    print(number)
devuelve un num del 1 al 10

Shuffle -> randomiza el orden de la seq

ej: cards = [“jack, “queen”, “king”]
    random.shuffle(cards)
    for cards in cards:
        print(card)
devuelve la lista randomizada, ej: [“queen”, “king”, “jack”]

# Statistics

Mean -> da la mediana

ej:
print(statistics.mean( [100, 90] )
da la mediana, osea 95

# PACKAGES

# PyPi

->Python Package index, allows us to download and install all sort of packages.

# Pip

-> allows us to install almost everything in cmd

# Cowsay

-> a package that. Once installed using “pip install cowsay” we can import ir as a normal library inside our code.

ej:

```
impor cowsay
import sys

if len(sys.argv) == 2:
        cowsay.cow(“hello, “ + sys.argv[1])
```

we run: $ python say.py David

nos devuelve una vaca dibujada con un globo diciendo “hello, david”

# APIs

-> Application Programming Interface. Refers to 3rd party services.

## Requests

-> pide las apis que queramos. Utiliza a veces JSON (Javascript Object Notation), JSON se utiliza como un “language agnostic (we dont need to use js) format for exchanging data between computers”

ej:

```
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response= requests.get(“https://itunes.apple.com/search?entity=song&limit=1&term=” + sys.argv[1])

print(response.json())
```

Terminal: python itunes.py weezer
devuelve un diccionario de python con toda la data de weezer (una banda o algo asi).

# JSON

lo que va a hacer json es formatear mi data (el diccionario gigante que recibimos) un poco mas limpio. Entonces en el codigo anterior:

```
import requests
import sys

if len(sys.argv) != 2:
        sys.exit()

response=
requests.get("https://itunes.apple.com/search?entity=song&limit=1&term=" + sys.argv[1])

print(json.dumps(response.json(), indent=2))
```

Terminal: python itunes.py weezer
devuelve lo mismo pero mas ordenado.

Ahora queremos que devuelva todas las canciones de la banda Weezer, como se hace esto?:

```
import requests
import sys

if len(sys.argv) != 2:
        sys.exit()
```

response=
requests.get("https://itunes.apple.com/search?entity=song&limit=50&term=" + sys.argv[1])

print(json.dumps(response.json(), indent=2))

**// buscamos trackName en el json de**
**//respuesta y accedemos a eso**
**o = response.json()**
**for result in o["results"]:**
**print(result["trackName"])**

terminal: python itunes.py weezer
nos devuelve 50 canciones que tiene Weezer (recordar que para que devuelva 50 canciones, tenemos que cambiar a limit = 50 en el link.

# Custom Libraries

Con python tambien podemos hacernos nuestras propias librerias
Lo podemos ver en la carpeta miLib que hice.

# UNIT TESTS

## Assert

-> me pide que algo se cumpla, por ej:

```
def test_square():
assert square(2) == 4
```

aca pedimos que el cuadrado de 2 sea 4, es mas rapido y simple de escribir.

## Try and Except

-> Si tenemos solo un assert, al haber un error, el cmd no nos va a decir claramente donde se encuentra y que tipo de error es.
Para esto usamos un try y un except:

- Try nos pide de intentar un test
- Except nos dice que hacer si no cumple

```
def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared was not 4")
```

Para muchos tests, va a ser recontra engorroso, nos va a quedar un codigo de muchisimas lineas para testear diferentes tests.
Para esto tenemos **pytest**

# Pytest

-> Es un 3rd party program que se puede descargar e instalar, y que automatizara los testings del codigo, siempre y cuando escribamos los tests.
Para correr un test en cmd es:
**pytest nombre_del_test.py**

Esto nos devolvera cuantos tests paso, cuantos no y el error.

Para simplificar el testeo, se puede (y se debiera) crear una carpeta **“test”** que contenga adentro el **test_del_programa.py** y **__init__.py**.

No importa si el __init__ esta vacio, pues sirve como indicador visual para python de que deberia usar esa carpeta como un **package** (un module de Python organizados dentro de una carpeta).

Con esto puedo directamente hacer:
**pytest test**
y me corre todos los tests que estan dentro de la carpeta “test”

# FILES I/O

## List

-> se inicializa asi “ names = [ ] “
ej:

```
names = []

for _ in range (3):
        names.append(input(“Whats your name? “))
for name in sorted(names):
        print(f”hello, {name}”)
```

devuelve:
nos pide 3 nombres distintos, y luego los printeo diciendo “hello, nombres”

Nos gustaria guardar la informacion, para eso tenemos **File I/O**

### Open

-> el equivalente para el programador de doble clicking en un icon en la pc. Nos permitira especificar exactamente que queremos que lea desde o escribe en esa file.

ej:
file = open(“names.txt”, “w”)
como quiero tambien escribir en esa file, voy a escribir “w” (write)

names.py

```
name = input(“Whats your name? “)

file = open(“names.txt”, “w”) # abro un nuevo file txt
file.write(name)              # escribo el nombre q puso el usuario
file.close()                  # cierro
```

Terminal:
python names.py
Whats your name? Ron
Whats your name? Hermione
code names.txt  # me abre el txt que ahora contiene Hermione

Solo guarda 1 valor, porque no estamos appending.

si hago:

names.py

```
name = input(“Whats your name? “)

file = open(“names.txt”, “a”) # abro un nuevo file txt
file.write(name)              # escribo el nombre q puso el usuario
file.close()                  # cierro
```

**a** -> append

Terminal:
python names.py
Whats your name? Ron
Whats your name? Hermione
Whats your name? Harry

code names.txt  # me abre el txt que ahora contiene:
RonHermioneHarry

¡Qué feo! Como lo arreglo?
Queremos agregar una linea en cada nombre

names.py

```
name = input("Whats your name? ")

file = open("names.txt", "a") # abro un nuevo file txt
file.write(f"{name}\n")       # escribo el nombre q puso el usuario
file.close()                  # cierro
```

me devuelve:
**Ron**
**Hermione**
**Harry**

No estrictamente debemos hacer file.close() para terminar con el file, es mas, esto nos puede llegar a dar problemas en el futuro. Parta evitar esto, vamos a tomar otro approach usando la palabra **with**

## With

-> permite abrir y cerrar automáticamente una file. Veamos:

names.py

```
name = input(“Whats your name? “)

# abro, edito y cierro dentro de with
with open(“names.txt”, “a”) as file:
        file.write(f”{name}\n”)    # escribo el nombre q puso el usuario
        file.close()               # cierro
```

Pero ahora quiero leer mi file, como hago esto?

```
with open(“names.txt”, “r”) as file:
        for line in file:
                print(“hello,”, line.rstrip())
```

Que pasaria si ademas de los nombres tambien queremos agregar sus casas?

si tuvieramos algo asi:

```
Hermione
Gryffindor
Harry
Gryffindor
Ron
Gryffindor
Draco
Slytherin
```

no seria muy claro, podemos confundirnos facilmente

Es por esto que muchos programadores utilizan archivos .csv
(Comma-separated Values)

Entonces creamos “students.csv”
y separaremos las casas con una coma:

students.csv

Hermione,Gryffindor
Harry,Gryffindor
Ron,Gryffindor
Draco,Slytherin

Ahora creamos un students.py para leer el students.csv:

```
with open(“students.csv”) as file:
        for line in file:
                row = line.rstrip().split(“,”)
                # rstrip -> removes the white space at the end
                # split(“,”) -> splitea en 2 al encontrar una , (coma)

                print(f” {row[0] is in {row[1]} ”) # printea los 2 splits
```

devuelve->
Hermione is in Gryffindor
Harry is in Gryffindor
Ron is in Gryffindor
Draco is in Slytherin

Se puede escribir mas lindo de esta manera:

```
with open(“students.csv”) as file:
        for line in file:
                name, house = line.rstrip().split(“,”)
                # el 1er split sera el name y el 2do la casa

                print(f” {name} is in {house} ”) # printea los 2 splits
```

Supongamos que queremos ordenar nuestros outputs basandonos en el nombre de mis estudiantes. Como hacemos?

# Dictionaries

Crearemos un diccionario que almacene los nombres y casas como keys:

codigo:

```
students = []

with open(“students.csv”) as file:
        for line in file:
                name, house = line.rstrip().split(“,”)
                student = {}       #{} -> diccionario vacio
                student[“name”] = name
                student[“house”] = house
                student.append(student)

def get_name(student):
        return student[“name”]
```

```
for student in sorted(students, key = get_name):
        print(f"{student['name']} is in {student['house'] "]} “)
        # necesitamos usar ' ' porque ya estoy usando “ ” afuera
```

Podemos simplificarlo aun mas al eliminar el get_name y escribir una **lambda** function (lambda es una funcion anonima, que no tiene nombre)

```
students = []

with open(“students.csv”) as file:
        for line in file:
                name, house = line.rstrip().split(“,”)
                student = {}        #{} -> diccionario vacio
                student[“name”] = name
                student[“house”] = house
                student.append(student)

for student in sorted(students, key = lambda student: student[“name”]):
        print(f”{student['name']} is in {student['house'] "]} “)
```

# Csv

Supongamos que queremos poner los datos de donde viven Draco, Ron y Harry:

Draco, Malfoy Manor
Ron, The Burrow
Harry, Number 4, Privet Drive

El hogar de Harry, al tener 1 coma va a tirar error en nuestro codigo actual. Para solucionar esto vamos a utilizar la libreria de csv.

# Reader

-> pertenece a csv. lee un csv file por vos y realiza donde estan las comas, las quotes, otras cosas y las resuelve por vos.

codigo:

```
import csv

students =[]

with open(“student.csv”) as file:
        reader = csv.reader(file)
        for name, home in reader:
                students.append( { “name”: name, “home”: home } )

for student in sorted(students, key = lambda student:
student[“name”]):
        print(f”{student['name']} is in {student['home'] "]} “)
```

Con esto solucionamos nuestro problema con el caso de Harry.

# DictReader

En vez de leerlo como lista con Reader, lo leeremos como un diccionario.
Primero de todo, en el students.csv que teniamos:
Draco, Malfoy Manor

Ron, The Burrow
Harry, Number 4, Privet Drive

Vamos a tener que agregarle los headers:

**name, home**
Draco, Malfoy Manor
Ron, The Burrow
Harry, Number 4, Privet Drive

Luego el codigo queda casi intacto, salvo por el hecho de que vamos a tener que utilizar row en vez de name, home pues es un diccionario.

```
import csv

students =[]

with open(“student.csv”) as file:
        reader = csv.DictReader(file)
        for row in reader:
                students.append( { “name”: row[“name”], “home”: row[“home”] } )

for student in sorted(students, key = lambda student: student[“name”]):
        print(f"{student['name']} is in {student['home']} ")
```

Veamos como seria el codigo para ingresar valores al diccionario:

```
import csv

name = input(“Whats your name?”)
name = input(“Wheres your home?”)

with open(“students.csv”, “a”) as file: #a -> append
        writer = csv.DictWriter(file, fieldnames =[“name”, “home”])
        writer.writerow({“name”: name, “home” : home})

        #fieldname -> los encabezados de mi dicc
```

# PIL (pillow)

-> libreria de python que permite navegar y llevar a cabo image files.

# Regular Expressions (Regex)

Es un patron que matchea algun tipo de data, generalmente el input del usuario.
Por ej, si el usuario tipea en un email address, ya sea en tu programa o en la website, o una app del celular, queremos validar que realmente tipear un email address

Supongamos que tenemos este codigo que dice si un mail es correcto o no:

```
email = input(“Whats your email? “).strip()

username, domain = email.split(“@”)

if username and domain.endswith(“.edu”):
        print(“valid”)
else:
        print(“Invalid”)
```

-> si elijo matias@.edu es valido, pero no deberia serlo

Para no complicarnos las vida, tenemos una libreria de regex que nos va a ayudar

# re

-> Sirve para definir y chequear replacing patterns.
Recordemos que un regex es un patron. Esta libreria nos va a dejar definir algunos de esos patrones.

# Search

**-> re.search(pattern, string, flags = 0)**

- pattern -> el patron que queramos buscar
- string -> el string que queremos buscar en el patron
- flag -> parametro que se le puede pasar para modificar el comportamiento de la funcion.

El pattern puede tomar una diferente cantidad de special symbols:

- . -> any character
- * -> 0 or more repetitions
- + -> 1 or more repetitions
- ? -> 0 or 1 repetitions
- {m} -> m repetitions
- {m,n} -> m-n repetitions

Veamos como mejorar mi anterior codigo:

```
import re

email = input("Whats your email? ").strip()

# con +, pedimos que hayan 1 o mas caracteres antes y despues del @
# con ".", pedimos que los caracteres pueden ser cualesquiera
if re.search(".+@.+", email):
```

```
    print("valid")
else:
    print("invalid")
```

Ahora, si queremos que termine en .edu, usaremos dentro del string de search un "\" para salir del mundo regex y una "r" que indica una raw string para que no trate de interpretar \ como regex:

```
import re

email = input("Whats your email? ").strip()

if re.search(r".+@.+ \.edu", email):
    print("valid")
else:
    print("invalid")
```

si ingresamos un input: mi email es matiasgangui@gmail.edu
Obtenemos que es invalido. Esto esta mal.

Introduzcamos mas simbolos:

^ -> **matchea el comienzo del string**
$ -> **matchea el final del string**, o justo antes de la newline al final de la string

Usemosla:

```
import re
email = input("Whats your email? ").strip()
if re.search(r"^.+@.+ \.edu$", email):
    print("valid")
```

```
else:
        print(“invalid”)
```

Ahora, si ingresamos un input: mi email es matiasgangui@gmail.edu
Obtenemos que es valido

pero todavia tenemos este problema:
input -> matias@@gmail.edu nos da Valid

Introducimos nuevos simbolos:

[ ] -> **set of characters**
[^] -> **complementando el set, no podes matchear ninguno de estos characters.**

veamos:

```
import re
email = input(“Whats your email? “).strip()
if re.search(r“^[^@].+@[^@]+ \.edu$”, email):
        print(“valid”)
else:
        print(“invalid”)
```

#Quitamos el . de la derecha del @ y escribimos de los dos lados [^@], que significa que podemos ingresar cualquier caracter que no sea @. Con esto estamos prohibiendole al usuario ingresar mas de un @.

Recapitulemos que significa todo este bolonqui de regex:
r“^[^@].+@[^@]+ \.edu$”

La expresión regular comprueba que la cadena comienza con cualquier carácter que no sea "@" (ya que se especifica con [^@] después del ancla ^), seguido de uno o más caracteres (cualquier carácter, excepto un salto de línea), seguido de un símbolo "@", seguido de uno o más caracteres (cualquier carácter, excepto "@") que representan el dominio, seguido de la cadena literal ".edu".

Todavia no esta totalmente correcto, porque si ingresamos .edu@gmail.edu , este esta correcto, pero no deberia porque hay un simbolo (.).

Para esto, en vez de restringir el uso del @ al principio del mail ([^@]) podemos directamente pedir que solo se ingresen letras **([a-z])** minsculas, **([A-Z])** mayusculas, numeros **([0-9])**, undescores **( _ )**. con lo que nos quedaria todo junto: **[a-zA-Z0-9_]**

```
import re
email = input(“Whats your email? “).strip()
if re.search(r“^[a-zA-Z0-9_].+@[a-zA-Z0-9_]+ \.edu$”, email):
        print(“valid”)
else:
        print(“invalid”)
```

Ahora si testeamos .edu@gmail.edu , nos da invalid
mati_gangui888@gmail.edu es valid
mati@@gmail.edu es valid

Esto que pusimos recien ([a-zA-Z0-9_]) se puede reemplazar por un shortcut (\w) que representa un “word character”

```
import re
email = input(“Whats your email? “).strip()
if re.search(r“^\w.+@\w+ \.edu$”, email):
        print(“valid”)
else:
        print(“invalid”)
```

Que pasa si ingreso MATIASG@GMAIL.EDU ?
Nos da incorrecto, esto puede estar bien, pero es mejor que logremos adaptar el mail ingresado a uno aceptable.

Recordemos el operador search:
re.search(pattern, string, flags = 0)
flags tiene opciones de configuracion que me permite configurar una funcion un tanto diferente, usemosla para resolver este problema.
Las flags que podemos pasar son estas:

re.IGNORECASE -> **ignora las cases, osea cualquier tipo de caracter es valido**
re.MULTILINE -> **para muchas lineas**
re.DOTALL -> **configura el dot (.) para reconocer no solo todos los caracteres excepto newlines, sino tambien cualquier caracter y newlines.**

Probemos la 1ra:

```
import re
email = input(“Whats your email? “).strip()
if re.search(r“^\w.+@\w+ \.edu$”, email, re.IGNORECASE):
        print(“valid”)
else:
        print(“invalid”)
```

MATIASG@GMAIL.EDU -> nos devuelve valid woohooo

Ahora, que pasa si quiero ingresar matias@cs50.harvard.edu?
-> invalid, auch, que ganas de morir no?

Por que? Porque hay un punto (.) que no estoy aceptando con \w

Veamos nuevas herramientas:

- A | B -> **either A or B**
- (...) -> **a group**
- (?:...) -> **non-capturing version**

Queremos decirle al regex, que parte del pattern es opcional, y que podemos ponerle 1 extra caracter + punto a mi mail:

```
import re
email = input("Whats your email? ").strip()
if re.search(r"^\w.+@(\w+\.)?\w+\.edu$", email, re.IGNORECASE):
        print("valid")
else:
        print("invalid")
```

#Con esto logramos que el usuario pueda ingresar 1 extra subdomain, pedimos que ingrese un caracter (\w) y un punto (\.) y luego nos fijamos si hay 0 o 1 repeticion (?)

Entonces ahora, matias@cs50.gmail.edu es valido

# Match, Fullmatch

**-> re.match(pattern, string, flags = 0)**
No necesito especificar el simbolo ^ al principio de mi regex si quiero matchear desde el principio de un string.
re.match, por definicion, automaticamente matcheara del principio del string por mi.

**-> re.fullmatch(pattern, string, flags = 0)**
Similar a match, pero match tambien al final del string para evitar escribir el simbolo ^ o $

---

Volvamos al ejercicio anterior, sobre el gmail, y veamos como hacer para, en vez de validar el input del usuario y ver si es correcto o no, asumamos que el usuario no va a escribir como queremos, y que tenemos que “limpiar” su input.

---

Creemos un nuevo .py, e intentemos corregir el users name solamente. Que tiene que ser “nombre apellido” (ej: Matias Gangui)

format.py

```
import re

name = input(“Whats your name? “).strip()
if matches := re.search(r”^(.+), *(.+)$”, name):
        name = matches.group(2) + “ “ + matches.group(1)

print(f”hello, {name}”)
```

:= -> **permite asignar una value y hacer una pregunta booleana**

# Sub

-> re.sub(pattern, repl, string, count = 0, flags = 0)

- pattern -> **the regex ta we are looking for**
- repl -> **replacement string, what do we want to replace that pattern with, and where do we want to do that.**
- count -> **how many times we want to do find and replace.**
- flags -> **parametro que se le puede pasar para modificar el comportamiento de la funcion.**

Veamos otro ejercicio, creemos un twitter.py, el objetivo es promptear users por el URL de su twitter profile y extraer de él, infiriendo de la url, cual es el username.
Por ej, dado https://twitter.com/matiasgangui
Inferir el user, osea matiasgangui
Para esto tenemos que aceptar un buen url, entonces tenemos que definir su comportamiento en regex y obtener la ultima parte, que es el user

twitter.py

```
import re

url = input("URL: ").strip()

#substituye la url por solo el user
username = re.sub(r"^(https?://)?(www\.)?twitter\.com/", "", url)
print(f"Username: {username}")

#https? tambien se puede poner (http|https)
```

Que pasa si ingresamos: https://www.google.com/ ?
-> nos devuelve https://www.google.com/. Esta mal

Veamos como solucionar esto:

twitter.py

```
import re

url = input("URL: ").strip()
if matches := re.search(r"^https?://(www\.)?twitter\.com/(.+)$", url, re.IGNORECASE)
    print(f"Username: ", matches.group(2))
    #group(1) = (www\.); group(2) = (.+)
```

Vemos que en este caso, (www\.) se toma como group(1), y al menos aca no molesta mucho, pero en regexs mas grandes puede llegar a sernos un problema contar la cantidad de groups.
Por eso recordamos que tenemos **(?:...) -> non-capturing version.**

Lo que significa que si escribimos (?:www\.), este no se va a tener en cuenta para el group. Y entonces nos quedaria que (.+) es group(1). Veamos como quedaria:

twitter.py

```
import re

url = input("URL: ").strip()
if matches := re.search(r"^https?://(?:www\.)?twitter\.com/(.+)$", url, re.IGNORECASE)
    print(f"Username: ", matches.group(1))
```

Hagamos un cambio final, estamos aceptando uno o mas caracteres. Twitter solo admite letras del alfabeto, numeros y underscores.

twitter.py

```
import re

url = input("URL: ").strip()
if matches :=
re.search(r"^https?://(?:www\.)?twitter\.com/([a-z0-9_]+)", url,
re.IGNORECASE)
        print(f"Username: ", matches.group(1))
#Cambiamos el "." por las caracteristicas que son validas.
```

# Split

**-> re.split(pattern, string, maxsplit = 0, flags = 0)**
Permite splitear un string, sin usar un caracter especifico o caracteres como una coma y espacio, pero varios caracteres tambien.

# Findall

**-> re.findall(pattern, string, flags = 0)**
Te permiten buscar múltiples copias del mismo patron en diferentes lugares dentro de una string asi tal vez se pueda manipular mas de solo una.

—----------

Symbols resume

- . -> any character
- * -> 0 or more repetitions
- + -> 1 or more repetitions
- ? -> 0 or 1 repetitions
- {m} -> m repetitions
- {m,n} -> m-n repetitions
-

^ -> **matchea el comienzo del string**

$ -> **matchea el final del string**, o justo antes de la newline al final de la string

[ ] -> **set of characters**
[^] -> **complementando el set, no podes matchear ninguno de estos characters.**

\d -> **decimal digit**
\D -> **not decimal digit**
\s -> **whitespace characters (espacio)**
\S -> **not a whitespace character**
\w -> **word character… aswell as numbers and the underscore**
\W -> **not a word character**

**Flags—----------**
re.IGNORECASE -> **ignora las cases, osea cualquier tipo de caracter es valido**
re.MULTILINE -> **para muchas lineas**
re.DOTALL -> **configura el dot (.) para reconocer no solo todos los caracteres excepto newlines, sino tambien cualquier caracter y newlines.**
—----------

- A | B -> **either A or B**
- (...) -> **a group**
- (?:...) -> **non-capturing version**

—----------

# OOP
# (OBJECT ORIENTED PROGRAMMING)

Empecemos creando un programa que pregunte al user or su nombre, su casa y que printee de donde es el user:

student.py

```
name = input(“Name: “)
house = input(“House: “)
print(f”{name} from {house}“)
```

Funciona perfecto, pero modifiquemos un poco:

```
def main():
```

```
    name = get_name()
    house = get_house()
    print(f"{name} from {house}")

def get_name():
    return input("Name: ")

def get_house():
    return input("House: ")

if __name__ == "__main__"
    main()
```

Sigue funcionando, sigamos modificando

```
def main():
    name, house = get_student()
    print(f"{name} from {house}")

#simplifiquemos get_house y get_student en un def
def get_student():
    name = input("Name: ")
    house = input("House: ")
    return name, house

if __name__ == "__main__"
    main()
```

Acabamos de usar una **tupla**

# Tuple

-> en return name, house no estoy realmente 2 values, sino que estoy retornando una tupla con 2 valores adentro.
Sabiendo esto, podemos modificar el codigo nuevamente:

```
def main():
        student = get_student()
        print(f"{student[0]} from {student[1]}")

def get_student():
        name = input("Name: ")
        house = input("House: ")
        return (name, house)

if __name__ == "__main__"
        main()
```

**Cuando usar una tupla?**
Cuando queremos programar de forma defensiva (wtf?), o , en general, cuando sabemos que las values de la variable no debieran cambiar. Si esto pasara no podriamos usar una lista por ej, porque esta puede ser cambiada. En una tupla nadie, ni el programador mismo, puede cambiar su contenido (**es inmutable)**.

Ahora, volviendo al codigo, si ingresamos name: Padma, house: gryffindor, obtenemos que "Padma from Gryffindor". Pero eso no es cierto, pues Padma es de Ravenclaw.
Veamos como corregirlo:

```
def main():
        student = get_student()
        if student[0] == “Padma”:
                student[1] = “Ravenclaw”

        print(f”{student[0]} from {student[1]}“)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
        return (name, house)

if __name__ == “__main__”
        main()
```

Funciona?
No, hay un TypeError: “tuple” object does not support item assignment

# Lists

Notamos que si queremos override el codigo con el caso de Padma, necesariamente vamos a tener que utilizar algo que no sea inmutable como las tuplas, podriamos intentar con lista:

```
def main():
        student = get_student()
        if student[0] == “Padma”:
                student[1] = “Ravenclaw”
        print(f”{student[0]} from {student[1]}“)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
```

```
    return [name, house]  # estamos devolviendo una lista

if __name__ == “__main__”
    main()
```

Ahora si, al usar una lista (que es mutable) funciona perfectamente el caso de Padma.

Veamos otra manera, esta vez con **Diccionarios**

# Diccionarios

```
def main():
    student = get_student()
    print(f”{student['name'} from {student['house'}“)

def get_student():
    student = { }
    student[“name”] = input(“Name: “)
    student[“house”] = input(“House: “)
    return student

if __name__ == “__main__”
    main()
```

podemos simplificarlo:

```
def main():
        student = get_student()
        print(f"{student['name'} from {student['house'}")

def get_student():
        student = { }
        name= input("Name: ")
        house = input("House: ")
        return {"name": name, "house": house}

if __name__ == "__main__"
        main()
```

Los diccionarios, como las listas, son **mutables**. Veamos como hacer el caso de Padma en Ravenclaw:

```
def main():
        student = get_student()
        if student["name"] == "Padma:
                student["house"] = "Ravenclaw"
        print(f"{student['name'} from {student['house'}")

def get_student():
        student = { }
        name= input("Name: ")
        house = input("House: ")
        return {"name": name, "house": house}

if __name__ == "__main__"
        main()
```

Todo esto es hermoso y lindo, pero nos podriamos haber ahorrado el usar diferentes tipos de estructuras si directamente los creadores de python implementaban un tipo de variable llamado student (que irresponsables!!). Es por eso, que en vez de crear este tipo de variable, nos dieron las herramientas para que lo creemos nosotros:

# Classes

Es una blueprint para pedazos de data. Es un molde en el que podemos definir tipos de data exactamente como queremos nosotros.
**En resumen, nos permite inventar nuestros propios tipos de data en python, y darles un nombre.**
Esto es una feature primordial en OOP.
Resolvamos el programa anterior usando classes:

```
class Student:
        … #veremos como implementarlo mas adelante

def main():
        student = get_student()
        print(f"{student.name} from {student.house}")

def get_student():
        student = Student() #Llamamos a la funcion class
        student.name = input("Name: ")
        student.house = input("House: ")
        return student

if __name__ == "__main__"
        main()
```

En student.name, student.house, creamos 2 nuevas variables de mi class Student que van a abarcar el nombre y la casa.

Sin implementar absolutamente nada en class (...), vemos que el codigo funciona perfectamente. Es que en python, ya con crear la class, es suficiente para que por ahora funcione.
En student = Student(), lo que estoy haciendo es crear un objeto de esa clase.
Si volvemos a la teoria de que una class es un molde, entonces un objeto es cuando usamos ese molde para crear una instancia especifica de la class.
Las clases son en default **mutables** pero podemos convertirlas a **inmutables**, luego veremos como.

Modifiquemos un poco mas el codigo:

```
class Student:
        …  #veremos como implementarlo mas adelante

def main():
        student = get_student()
        print(f"{student.name} from {student.house}“)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
        student = Student(name, house)
        return student

if __name__ == “__main__”
        main()
```

#Les paso argumento a la class

Ahora tendremos que ver como tomarlo dentro de la class Student

# Methods

Las clases vienen con ciertos methods, o funciones dentro suyas, que podemos definir y se comportan de una manera especial.
Estas funciones nos permiten determinar el comportamiento de una manera estandar.

```
class Student:
        def __init__(self, name, house):
                #agregamos variables a objetos
                self.name = name
                self.house = house

def main():
        student = get_student()
        print(f"{student.name} from {student.house}“)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
        student = Student(name, house) #constructor call
        return student

if __name__ == “__main__”
        main()
```

La manera de inicializar el contenido de un objeto desde una clase, tenemos que definir un __init__.
La funcion que va a ser siempre llamada, por definicion de las clases de python, es __init__. Es lo que los autores de python eligieron para implementar la inicializacion de un objeto en Python.
El “self” dentro de __init__(self, name, house) tiene que estar para guardar los valores de name y house. Es decir que **self siempre va a estar en __init__ y siempre esta primero, y el self nos da acceso a los objetos que fueron creados.**

Como nos cercioramos que el user input ponga casas validas?
Classes!! chachaaan!!
Si queremos validar que una casa es correcta o no, podemos usar:

# Raise

-> permite crear tus propias excepciones cuando algo sale mal, permite alertar al programador.

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Missing name")

        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
        "Slytherin"]:
            raise ValueError("Invalid house")

        self.name = name
        self.house = house

def main():
    student = get_student()
    print(f"{student.name} from {student.house}")

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__"
    main()
```

# __init__

Es un special method que si lo definis dentro de tu clase, python automaticamente va a llamarla por vos cuando alguna funcion quiera ver tu objeto como string.
Definamosla en la class del codigo:

```
class Student:
    def __init__(self, name, house):
        if not name:
            raise ValueError("Missing name")

        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
        "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

def main():
    student = get_student()
    print(Student) #print va a pedir un __str__

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)

if __name__ == "__main__"
    main()
```

Ahora, porque nos pintó, vamos a agregar un nuevo componente. Patronus:

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")

        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
        "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"

def main():
    student = get_student()
    print(Student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus  = input("Patronus: ")
    return Student(name, house)

if __name__ == "__main__"
    main()
```

Ahora me pintó darle charms (hechizos) a los estudiantes. Implementarlo en la class:

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")

        if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw",
        "Slytherin"]:
            raise ValueError("Invalid house")
        self.name = name
        self.house = house
        self.patronus = patronus

    def __str__(self):
        return f"{self.name} from {self.house}"

    def charm(self):
        match self.patronus:
            case "Stag":
                return "🐎"
            case "Otter":
                return "🦦"
            case _:
                return "no patronus"

def main():
    student = get_student()
    print("Expecto Patronum!")
    print(student.charm())

def get_student():
    name = input("Name: ")
    house = input("House: ")
    patronus  = input("Patronus: ")
    return Student(name, house)
```

```
if __name__ == “__main__”
    main()
```

Bueno, removamos toda la parte de emojis y patronuses y enfoquemonos en otra capacidad que tienen las classes.

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError(“Missing name”)

        if house not in [“Gryffindor”, “Hufflepuff”, “Ravenclaw”,
        “Slytherin”]:
            raise ValueError(“Invalid house”)

        self.name = name
        self.house = house

    def __str__(self):
        return f”{self.name} from {self.house}”

def main():
    student = get_student()
    print(student)

def get_student():
    name = input(“Name: “)
    house = input(“House: “)
    return Student(name, house)
```

# Properties

-> Es un atributo que tiene mas mecanismos de defensa puestos en practica, un poco mas de funcionamiento que podemos implementar para que programadores no ensucien nuestro codigo. En resumen, para tener mas control sobre nuestro codigo.
Veremos **@property**,
**decorators** (funciones que modifican el comportamiento de otras funciones)**.**
**Veamos una implementacion:**

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    #Getter (funcion de una clase que obtiene atributos)
    @property
    def house(self):
        return self._house
    # tiene que ser _house xq house normal ya la estamos usando

    #Setter (funcion de una clase que setea algunos valores)
    @house.setter
        def house(self, house):
            if house not in ["Gryffindor", "Hufflepuff", "Ravenclaw", "Slytherin"]:
```

```
                raise ValueError("Invalid house")
            self._house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)
```

Testeemos si funciona, como? Pues asignando en el main un house que no pertenezca a ninguna de las casas que seteamos como validas:

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    #Getter
    @property
    def house(self):
        return self._house
```

```
        #Setter
        @house.setter
                def house(self, house):
                        if house not in [“Gryffindor”, “Hufflepuff”,
“Ravenclaw”, “Slytherin”]:
                                raise ValueError(“Invalid house”)
                        self._house = house

def main():
        student = get_student()
        student.house = “Number Four, Privet Drive”
        print(student)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
        return Student(name, house)
```

Como “Number Four, Privet Drive” no pertenece a ninguna de las casas que definimos, y como tenemos un getter que nos pide que usemos el setter para definir los errores, al iniciar setter, y ver que esta casa no esta dentro de las posibilidades, nos va a tirar el error que hemos definido. Es decir, obtendremos un **ValueError: Invalid House**. Justo lo que queriamos

En resumen, usamos getter y setter para asegurarnos de que los programadores no puedan hacer cosas como la de recien y modificar completamente el input( en este caso cambiar el house a uno q no es, por suerte al tener getter y setter nos salvamos de estos errores).
Nos sirve para poder tener mas control sobre este objeto para asi poder confiar mas en él.
Usar un getter y setter realmente solo habilita python para automáticamente detectar cuando estas tratando de setear una value manualmente. Al hacer esto, python primero se fijara en el

setter si realmente el nuevo valor que se le esta asignando es correcto.

Por ahora definimos **@property** para **house**. Hagamos tambien una para **name**.

```
class Student:
    def __init__(self, name, house, patronus):
        if not name:
            raise ValueError("Missing name")
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if not name:
            raise ValueError("Mising name")
        self._name = name

    #Getter
    @property
    def house(self):
        return self._house

    #Setter
    @house.setter
        def house(self, house):
```

```
                if house not in [“Gryffindor”, “Hufflepuff”,
“Ravenclaw”, “Slytherin”]:
                        raise ValueError(“Invalid house”)
                self._house = house

def main():
    student = get_student()
    print(student)

def get_student():
    name = input(“Name: “)
    house = input(“House: “)
    return Student(name, house)
```

Y ahora podemos sacarnos de encima el “if not name” en **__init__**

```
class Student:
    def __init__(self, name, house, patronus):
        self.name = name
        self.house = house

    def __str__(self):
        return f”{self.name} from {self.house}”

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if not name:
            raise ValueError(“Mising name”)
        self._name = name
```

```
        #Getter
        @property
        def house(self):
                return self._house

        #Setter
        @house.setter
                def house(self, house):
                        if house not in [“Gryffindor”, “Hufflepuff”,
“Ravenclaw”, “Slytherin”]:
                                raise ValueError(“Invalid house”)
                        self._house = house

def main():
        student = get_student()
        print(student)

def get_student():
        name = input(“Name: “)
        house = input(“House: “)
        return Student(name, house)
```

Por ahora todos los metodos que vimos fueron **instance methods**. En realidad hay mas metodos que los que vimos. Es hora de crecer y ver los :

# Class Methods

->Al parecer, a veces no es realmente necesario asociar una funcion con objetos de una clase, sino que con uno mismo.

A veces queremos alguna funcionalidad, alguna accion que sea asociada con la misma class, sin importar cuales son los valores o variables instanciados del objeto especifico.
Por esto tenemos **@classmethod**
Veamos un ejemplo:

hat.py

```
class hat:
        def sort(self, name):
                print(name, “is in”, “some house”)

hat = Hat()
hat.sort(“Harry”)
```

si lo corremos nos devuelve: “Harry is in some house”
Por ahora todo bien, ahora hagamos que funcione mas aleatoriamente, es decir que elija una casa aleatoria para los alumnos.

```
import random

class hat:
        def __init__(self):
                self.houses = [“Gryffindor”, “Hufflepuff”, “Ravenclaw”, “Slytherin”]

        def sort(self, name):
                print(name, “is in”, random.choice(self.houses))

hat = Hat()
hat.sort(“Harry”)
```

mejoremos el diseño del sorting hat para que no tengamos que instanciar un sorting hat. Por que? porque podriamos pedir multiples sorting hats:
hat1 = Hat()
hat2 = Hat()
…
Como en realidad solo necesitamos un sorting hat (y ademas solo existe uno en harry potter), no necesitamos instanciar varios.

# @classmethod

-> Es un decorator que nos permite usar la class como un container de data. Veamos como lo implementamos:

```
import random

class hat:
    houses = [“Gryffindor”, “Hufflepuff”, “Ravenclaw”, “Slytherin”]

    def sort(self, name):
        print(name, “is in”, random.choice(self.houses))

hat = Hat()
hat.sort(“Harry”)
```

Eliminamos el __init__ porque solo servia para inicializar objetos especificos de la template, mold de la class.
Terminamos declarando houses como una variable. No es mas “self”, es una variable dentro de la class. Como ahora esta dentro de mi class, puedo usar la lista dentro de cualquiera de mis funciones.

Lo mismo podria hacerse con sort, no tiene mucho sentido sortear en un especifico sorting hat cuando realmente necesitamos un hat:

```
import random

class hat:
        houses = [“Gryffindor”, “Hufflepuff”, “Ravenclaw”, “Slytherin”]

        @classmethod
        def sort(cls, name):
                print(name, “is in”, random.choice(cls.houses))

hat = Hat()         #Esto lo podemos eliminar y voila!
Hat.sort(“Harry”)
```

—----------

Finalmente nos queda:

```
import random

class hat:
        houses = [“Gryffindor”, “Hufflepuff”, “Ravenclaw”, “Slytherin”]

        @classmethod
        def sort(cls, name):
                print(name, “is in”, random.choice(cls.houses))

Hat.sort(“Harry”)
```

—----------

Volvamos al student.py e implementemos todo lo que aprendimos recien:

```
class Student:
    def __init__(self, name, house, patronus):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

def main():
    student = get_student()
    print(student)

def get_student():
    name = input("Name: ")
    house = input("House: ")
    return Student(name, house)
```

Que podemos ver que podriamos mejorar?
En 1er lugar, tenemos una def get_student() fuera de la class. Es un poco raro, porque es una funcion que ayuda a student pero no esta en class.

```
class Student:
    def __init__(self, name, house, patronus):
        self.name = name
        self.house = house

    def __str__(self):
        return f"{self.name} from {self.house}"

    @classmethod
    def get(cls):
        name = input("Name: ")
        house = input("House: ")
```

```
            return cls(name, house)

def main():
        student = Student.get()
        print(student)
```

# Inheritance

-> Se puede diseñar las classes en una forma jerarquica, donde podemos tener una class que hereda o pide prestado de variables o metodos de otra clase.

Veamos como implementar esto, crearemos un wizard.py:

wizard.py

```
class Student:
        def __init__(self, name, house):
                if not name:
                        raise ValueError("Missing name")
                self.name = name
                self.house = house
                …

class Professor:
        def __init__(self, name, subject):
                if not name:
                        raise ValueError("Missing name")
                self.name = name
                sef.subject = subject
                …
```

Vamos viendo que las clases que vamos creando son redundantemente parecidas, y es una paja copiarlas siempre.
Para esto tenemos la **inheritance.**
Podemos definir muchas classes que mas o menos se relacionan entre ellas. Por ej, podemos decir que Student y Professor, en realidad son wizards. Entonces podriamos crear una tercera class que tiene los atributos en comun entre ellos.

```
class Wizard:
        def __init__(self, name):
                # referencia a superclass de esta class
                super().__init__(name)
                if not name:
                        raise ValueError("Missing name")
                self.name = name

class Student(Wizard): #Student hereda de Wizard
        def __init__(self, name, house):
                #Hereda toda la funcionalidad de Wizard, y pasamos los nombres de los Wizards a Student
                super().__init__(name)
                self.house = house
                …

class Professor(Wizard): #Professor hereda de Wizard
        def __init__(self, name, subject):
                #Hereda toda la funcionalidad de Wizard, y pasamos los nombres de los Wizards a Professor
                super().__init__(name)
                sef.subject = subject
                …
wizard = Wizard("Albus") #Creamos un wizard cualquiera
student = Student("Harry", "Gryffindor") #Creamos un student
professor = Professor("Severus", "Defense Against the Dark Arts")
```

# Operator Overloading

->Podemos declarar simbolos e implementarlos como otra cosa.
Por ej, pedir que el “+” escriba hello world y cosas asi.
Creamos la bodega (volt) de gringotts:

vault.py

```
class Vault:
    def __init__(self, galleons = 0, sickles = 0, knuts = 0)
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f”{self.galleons} Galleons, {self.sickles} Sickles,
{self.knuts} Knuts”

potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)
```

Intentemos combinar los contenidos de los 2 vaults.

```
class Vault:
    def __init__(self, galleons = 0, sickles = 0, knuts = 0)
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f”{self.galleons} Galleons, {self.sickles} Sickles,
{self.knuts} Knuts”
```

```
potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)

galleons = potter.galleons + weasley.galleons
sickles = potter.sickles + weasley.sickles
knuts = potter.knuts + weasley.knuts

total = Vault(galleons, sickles, knuts)
print(total)
```

Medio trabajoso, capaz hay una mejor manera, capaz **haciendo operator overload**.
Dentro de estos operator overloads tenemos la existencia de **object.__add__(self, other)**
Donde self es la parte izq de la suma y other es la derecha, por ejemplo 3 + 5 -> self + other.

```
class Vault:
    def __init__(self, galleons = 0, sickles = 0, knuts = 0)
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts

    def __str__(self):
        return f"{self.galleons} Galleons, {self.sickles} Sickles, {self.knuts} Knuts"

    def __add__(self, other):
        galleons = self.galleons + other.galleons
        sickles = self.sickles + other.sickles
        knuts = self.knuts + other.knuts
        return Vault(galleons, sickles, knuts)
```

```
potter = Vault(100, 50, 25)
print(potter)

weasley = Vault(25, 50, 100)
print(weasley)

total = potter + weasley
```

->Lo que va a pasar al tener declarado un __add__, es que cuando python vea el “potter + weasley”, va a ir directamente a la class y buscar un __add__ para entender como funciona.

# ETCETERA…

## Set

-> Un tipo de dato para filtrar duplicates. Un array en C++

ejemplo:

```
students = [
        {“name”: “Hermione”, “house”: “Gryffrindor”},
        {“name”: “Ron”, “house”: “Gryffrindor”},
        {“name”: “Draco”, “house”: “Slytherin”},
        {“name”: “Padma”, “house”: “Ravenclaw”},
```

```
houses = set()
for student in students:
        houses.add(student[“house”])

for house in sorted(houses):
        print(house)
```

-> me devuelve las casas sorteadas alfabeticamente y sin repetidos

# Global

->global variables son variables definidas que son locales a una funcion o funciones que los posicionamos al principio de mi documento .py. Veamos un ejemplo:

bank.py

```
balance = 0

def main():
        print(“Balance: “, balance)
        deposit(100)
        withdraw(50)
        print(“Balance: “, balance)

def deposit(n):
        balance += n

def withdraw(n):
        balance -= n

if __name__ == “__main__”:
        main()
```

Obtenemos un:
UnboundLocalError: local variable “balance” referenced before assignment

Esto pasa porque la funcion no es global, sino local. Si queremos cambiar una variable global, tiene que estar dentro de un def.
Si intentamos ponerla dentro de algun def, tampoco funcionara.

Por eso vamos a ver como solucionarlo con **Global**

bank.py

```
balance = 0

def main():
    print(“Balance: “, balance)
    deposit(100)
    withdraw(50)
    print(“Balance: “, balance)

def deposit(n):
    global balance
    balance += n

def withdraw(n):
    global balance
    balance -= n

if __name__ == “__main__”:
    main()
```

Ahora funciona joya

Usemos un enfoque mas de OOP para esto:

```
class Account:
        def __init__(self):
                self._balance = 0

        @property
        def balance(self):
                return self._balance

        def deposit(self, n):
                self._balance += n

        def withdraw(self, n):
                self._balance -= n

def main():
        account = Account()
        print("Balance: ", account.balance)
        account.deposit(100)
        account.withdraw(50)
        print("Balance: ", account.balance)

if __name__ == "__main__":
        main()
```

# Constant

-> algunos lenguajes nos permiten definir variables que osn **constantes**, es decir que una vez que le definimos un valor, no se les puede modificar. Veamos:

meows.py

```
for i in range(3):
        print(“meow”)
```

El 3 sale de la nada, como de magia. Es mas claro definirlo antes:

```
Meows = 3

for i in range(Meows):
        print(“meow”)
```

Veamos como hacerlo con OOP, osea classes:

```
class Cat:
        Meows = 3

        def meow(self):
                for _ in range(Cat.Meows):
                        print(“meow”)

cat = Cat()
cat.meow()
```

# Type Hints

-> Python es un lenguaje dinamicamente tipeado. Dinamicamente tipeado es cuando queres tipear un int, tenes que avisarle que vas a tipear un int.
Lo que ocurre con Python es diferente, porque **python es un lenguaje no muy fuertemente tipeado**. Podemos definir variables y no especificar que tipo de datos son. Python intenta averiguar por si solo el tipo.

Tenemos un programa que es muy popular para saber si mi codigo se esta adhiriendo bien a los tipos de datos que python infiere que estoy usando:

## Mypy

-> Chequea si mis variables estan realmente usando los tipos correctos. Veamos:

meows.py

```
def meow(n):
        for _ in range(n):
                print(“meow”)

number = input(“Number: “)
meow(number)
```

.>Nos devuelve: TypeError: “str” object cannot be interpreted as an integer

El problema es que la funcion input retorna un string o str, no un int. Como lo arreglamos?

```
def meow(n: int):
        for _ in range(n):
                print(“meow”)

number = input(“Number: “)
meow(number)
```

Con esto se arregla? Nop, mismo error
Veamos si podemos ayudarnos de **mypy**

Corremos: $ mypy meows.py
-> error: Argument 1 to “meow” has incompatible type “str”;
Gracias mypy!!
—----------

```
def meow(n: int):
        for _ in range(n):
                print(“meow”)

number: int = input(“Number: “)
meow(number)
```

Corremos: $ mypy meows.py
-> error: Incompatible types in assignment (expression has type “str”, variable has type “int”)

Que podemos hacer para resolverlo??
—----------

```
def meow(n: int):
        for _ in range(n):
                print(“meow”)

number: int = int(input(“Number: “))
meow(number)
```

Corremos: $ mypy meows.py
-> Success: no issues found in 1 source file

—----------

Escribamos el codigo de otra manera:

```
def meow(n: int):
        for _ in range(n):
                print(“meow”)

number: int = int(input(“Number: “))
meows: str = meow(number)
print(meows)
```

si lo corremos e ingresamos n = 3, obtenemos:

```
meow
meow
meow
None        #Esto porque en la ultima linea queremos que
            nos devuelva el valor del def, pero al no tener return, es
            None por default
```

Esto es lo mismo que decir:

```
def meow(n: int) -> None:  #return vale None
        for _ in range(n):
                print(“meow”)

number: int = int(input(“Number: “))
meows: str = meow(number)
print(meows)
```

Si ahora corro mypy de esto obtenemos:
-> error: “meow” does no return a value

Resolvamoslo:

```
def meow(n: int) -> str:
        return “meow\n” * n

number: int = int(input(“Number: “))
meows: str = meow(number)
print(meows, end=””)
```

Ahora funciona re chetou!!!

Despues de tantos maullidos, es hora de cambiar un poco de tema, y ver sobre looooooos:

# Docstrings

-> En definicion, es como documentar tus python files, iupi!
Esto se hace, no con “#”, sino con triple comilla doble (“ “ “).
La diferencia es que, cuando python ve “””, asume que es la documentación de la funcion.
Documentemos el codigo anterior:

```
def meow(n: int) -> str:
    """
    Meow n times.

    :param n: Number of times to meow
    :type n: int
    :raise TypeError: If n is not an int
    :return: A string of n meows, one per line
    :rtype: str
    """
    return "meow\n" * n

number: int = int(input("Number: "))
meows: str = meow(number)
print(meows, end="")
```

Veamos otra forma de escribirlo:

```
import sys

if len(sys.argv) == 1:
        print(“meow”)
else:
        print(“usage: meows.py”)
```

-> si escribimos en la terminal $ python meows.py, funciona y printea un solo meow. Si le ponemos algo mas, como:
$ python meows.py 3 , printea el else.

Juguemos con el input

```
import sys

if len(sys.argv) == 1:
        print(“meow”)

#si ponemos -n m, printea m veces, m siendo un int
elif len(sys.argv) == 3 and sys.argv[1] == “-n”:
        n = int(sys.argv[2])
        for _ in range(n):
                print(“meow”)

else:
        print(“usage: meows.py”)
```

```
$ python meows.py -n 3
->meow
  meow
  meow
```

Si hiciera $python meows.py -n 2 -> nos daria 2 meows, y asi siguiendo.

# Argparse

->Es una libreria que maneja este analisis de command line arguments por nosotros automaticamente, asi podemos programar la parte mas interesante.
Usemoslo en el codigo anterior:

```
import argparse

parser = argparse.ArgumentParser()                  #constructor
parser.add_argument("-n")          #method en el parser object
args = parser.parse_args()         #mirara el sys.argv por nosotros

for _ in range(int(args.n)):
        print("meow")
```

funciona siempre y cuando le pongamos un -n m, si no ponemos nada se rompe. Podemos ver ayuda haciendo:
**python3 meows.py -h (o –help), esto nos devuelve:**

```
usage: meows.py [-h] [-n N]

options:
   -h , –help show this help message and exit
   -n N                              #N significa q hay q tipear un numero
```

Mejoremos el codigo para que este -h ayude un poco mas:

```
import argparse

parser = argparse.ArgumentParser(description = “Meow like a cat”)
parser.add_argument(“-n”, help =”number of times to meow”)
args = parser.parse_args()

for _ in range(int(args.n)):
        print(“meow”)
```

**python3 meows.py -h** nos devolvera:

```
usage: meows.py [-h] [-n N]

Meow like a cat

options:
-h, –help show this help message and exit
```

agreguemos MAS!!

```
import argparse

parser = argparse.ArgumentParser(description = “Meow like a cat”)
parser.add_argument(“-n”, default = 1, help =”number of times to
meow”, type = int)
args = parser.parse_args()

for _ in range(int(args.n)):
        print(“meow”)
```

Si corro python meows.py, gracias al default = 1, me devolvera 1 “meow”.

Miremos otra feature:

# Unpacking

-> Antes de explicarlo, veamos un ejemplo. Queremos convertir galleons, sickles y knuts a knuts:

unpack.py

```
def total(galleons, sickles, knuts):
        return (galleons * 17 + sickles) * 29 + knuts

coins = [100, 50, 25]

print(total(coins[0], coins[1], coins[3]), “Knuts”)
```

Funciona bien, pero si la lista fuera aun mas grande, empezaria a ser realmente trabajoso insertar valor por valor. Para esto tenemos el **unpacking:**

```
def total(galleons, sickles, knuts):
        return (galleons * 17 + sickles) * 29 + knuts

coins = [100, 50, 25]

print(total(*coins), “Knuts”)

#Al usar un * antes de la lista, va a unpackearlo, es decir que
*coins = (coins[0], coins[1], coins[3])
#Esto no funciona con Set
```

—----------

Veamos otra variante de esto:

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

print(total(galleons = 100, sickles = 50, knuts = 25), “Knuts”)
```

Funciona.

—----------

Y si ahora, porque soy re denso, lo implementamos con un diccionario??

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

coins = {“galleons”: 100, “sickles”: 50, “knuts”: 50}

print(total(coins[“galleons”], coins[“sickles”], coins[“knuts”]), “Knuts”)
```

—----------

Recordar que **podemos unpackear diccionarios usando **** en vez de *:

```
def total(galleons, sickles, knuts):
    return (galleons * 17 + sickles) * 29 + knuts

coins = {“galleons”: 100, “sickles”: 50, “knuts”: 50}

print(total(**coins), “Knuts”)
```

Estos asteriscos no solo se usan en Unpacking, sino tambien en muchos lugares como indicador visual para cuando en una funcion misma se pueden recibir una variable suma de argumentos. Por ej podemos usar ***args, ****kwargs**. Veamos:

```
def f(*args, **kwargs):
        print("Positional:", args)

f(100, 50, 25)

#*args ->esta funcion toma alguna variable cantidad de argumentos posicionales
# **kwargs -> Quiero una cantidad de keyword arguments que sean named parameters que puedan ser llamados opcionalmente e individualmente por su propio nombre
```

Esto nos devuelve -> Positional: (100,50,25)

Y si le agregamos un valor?

```
def f(*args, **kwargs):
        print("Positional:", args)

f(100, 50, 25, 5)
```

Esto nos devuelve -> Positional: (100,50,25, 5)
Lo que antes nos decia que nos se podia, ahora se puede. Esta manera es mas flexible para estos casos.

Seamos como Jorge, más curiosos. Esta vez printeemos el named argument (kwargs):

—----------

```
def f(*args, **kwargs):
        print("Positional:", kwargs)

f(galleons = 100, sickles = 50, knuts = 25)
```

—----------

nos devuelve -> Named: {"galleons":100, … , … }
Osea el f pero en forma de diccionario.
Vemos que **kwargs** es un diccionario que contiene todos los argumentos que fueron pasados hacia mi función

Veamos una nueva estructura muy importante

# Map

-> Vimos un lado de python llamado **functional programming**. Donde las funciones tienden a no tener side effects, printear o cambios de estado globales. Sino que estan completamente auto-contenidos.
Veamos como funciona el map con un programaque nos grita el resultado:

yell.py

```
def main():
    yell("This is cs50")

def yell(phrase):
    print(phrase.upper())

if __name__ == "__main__":
    main()
```

---

Hagamoslo con lista:

```
def main():
    yell(["This", "is", "cs50"])

def yell(words):
    #creamos una nueva lista que terminara appending
    #cada una de las palabras pero en uppercase
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(uppercased)

if __name__ == "__main__":
```

```
    main()
```

me devuelve [“THIS”, “IS”, “CS50”], no es lo que queremos. Falto el * antes del uppercased en print:

```
def main():
    yell([“This”, “is”, “cs50”])

def yell(words):
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

if __name__ == “__main__”:
    main()
```

ahora si -> THIS IS CS50

Podemos hacer mejor che:

```
def main():
    yell(“This”, “is”, “cs50”)

def yell(*words):   #tomara un numero variable de palabras
                    #(funcionara como un args, solo que le pusimos
                    #words xq es mas lindo)
    uppercased = []
    for word in words:
        uppercased.append(word.upper())
    print(*uppercased)

if __name__ == “__main__”:
    main()
```

Por suerte python viene con la funcion **Map**, que su proposito es permitirte aplicar algunas funciones a cada elemento de una secuencia (como una lista)
**map(function, iterable, …)**

Reimplementemos la funcion:

```
def main():
        yell(“This”, “is”, “cs50”)

def yell(*words):
        uppercased = map(str.upper, words)
        print(*uppercased)

if __name__ == “__main__”:
        main()
```

#Lo que hara map aca es:

- Iterar sobre todas las palabras
- llamar str.upper a cada palabra que se itere
- y retornar una nueva lista con los resultados

Hay ootra manera para tambien solucionar este problema, que raro no?, esta se llama:

# List Comprehensions

-> **la habilidad de construir una lista facilmente** sin la necesidad de usar un for loop, llamar append, etc. Te hace esto en solo una linea, re vagos los de python.
Miremos el codigo:

```
def main():
        yell(“This”, “is”, “cs50”)

def yell(*words):
        uppercased = [word.upper() for word in words]
        print(*uppercased)

if __name__ == “__main__”:
        main()

#Tambien funciona con el if condition
#No me gusto para nada esta alternativa
```

—----------

Veamos otro codigo:

gryffindors.py

```
students = [
    {"name": "Hermione", "house": "Gryffindor"},
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Ron", "house": "Gryffindor"},
    {"name": "Draco", "house": "Slytherin"},
]

#Me devuelve solo los que son de Gryffindor
gryffindors = [student[“name”] for student in students if
student[“house”] == “Gryffindor”]

for gryffindor in sorted(gryffindors):
    print(gryffindor)
```

Funciona!! Chetoo!!
Nos faltan ver otras funciones muy utiles para estos casos. Una es:

# Filter

-> **filter(function, iterable)**
Se puede usar para obtener el mismo efecto pero con un approach mas funcional, veamos un ejemplo:

```
students = [
    {"name": "Hermione", "house": "Gryffindor"},
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Ron", "house": "Gryffindor"},
    {"name": "Draco", "house": "Slytherin"},
]

def is_gryffindor(s):
        return s[“house”] == “Gryffindor”
```

```
gryffindors = filter(is_gryffindor, students)

for gryffindor in sorted(gryffindor, key = lambda s: s[“name”]):
        print(gryffindor[“name”])

#Filter espera como primer parametro un booleano, luego “filtra” dependiendo del booleano.
```

Esto se puede simplificar eliminando el def is_gryffindor  y definirla en un lambda dentro del filter:

```
students = [
    {"name": "Hermione", "house": "Gryffindor"},
    {"name": "Harry", "house": "Gryffindor"},
    {"name": "Ron", "house": "Gryffindor"},
    {"name": "Draco", "house": "Slytherin"},
]


gryffindors = filter(lambda s: s[“house”] == “Gryffindor”, students)

for gryffindor in sorted(gryffindor, key = lambda s: s[“name”]):
        print(gryffindor[“name”])

#Es literal lo mismo, pero para acortar el codigo usamos el lambda para definir la funcion cortita dentro del filter y no gastarnos en escribir mas
```

—----------

Para resumir, tenemos 2 formas de resolver este ejercicio:

1) List Comprehension:
    a) Dentro de esa list vamos filtrando, incluyendo un if conditional.
2) Approach Funcional:

a) Usando un filter function, pasando en la función que quiero que haga las decisiones por mi, y luego incluir aquellos que sean True

# Dictionary Comprehensions

-> **Nos da la habilidad de crear un diccionario facilmente**, son la necesidad de pasar por la creacion de un diccionario vacio, un for loop, iterar sobre ese loop, insertar keys, etc.
Veamos:

gryffindors.py

```
students = [“Hermione”, “Harry”, “Ron”]

gryffindors = []

for student in students:
    gryffindors.append({“name”: student, “house”: “Gryffindor”})

print(gryffindors)
```

—----------

Veamos como mejorarlo:

```
students = [“Hermione”, “Harry”, “Ron”]

gryffindors = [{“name”: student, “house”: “Gryffindor”} for student in students]

print(gryffindors)

#Et voila!
```

Y si no quiero una lista de diccionarios, pero un gran diccionario?

```
students = [“Hermione”, “Harry”, “Ron”]

gryffindors = [{student: “Gryffindor” for student in students}

print(gryffindors)
```

#gryffindors va a crear una variable que va a ser un diccionario, cada key sera un nombre de estudiante, cada value va a ser Gryffindor

Esta vez me dara un gran diccionario, no una lista.

Fijemonos en otro problema, queremos numerar los estudiantes, del 1 al número máximo de estudiantes que definamos en una lista. ¿Cómo hacemos?

gryffindorsNum.py

```
students = [“Hermione”, “Harry”, “Ron”]

for student in students:
        print(student)
```

me devuelve:
Hermione
Harry
Ron  ,  pero no están numerados :(

Y si hacemos:

students = [“Hermione”, “Harry”, “Ron”]

```
for i in range(len(students)):
        print(i + 1, student[i])
```

Esto si me devuelve:
1 Hermione
2 Harry
3 Ron

Pero nos olvidamos que existe una propiedad de python que lo hace por nosotros, osea para que pensar?

# Enumerate

**->enumerate(iterable, start = 0)**
Como quedaria el codigo?:

students = [“Hermione”, “Harry”, “Ron”]

```
for i, student in enumerate(students):
        print(i+1, student)
```

Et voila!

Otra mini herramienta por si las moscas!!!

# Generators

-> La habilidad de generar values en python de funciones. Veamoslo en una funcion que implementa contar ovejas para dormir.

sleep.py

```
def main():
        n = int(input("Whats n? "))
        for i in range(n):
                print("🐑" * i)

if __name__ == "__main__":
        main()
```

este codigo funciona, implementemos otra manera creando una funcion llamada sheep

```
def main():
        n = int(input("Whats n? "))
        for i in range(n):
                print(sheep(i))

def sheep(n):
        return "🐑" * n

if __name__ == "__main__":
        main()
```

funca, pero es medio feo el codigo, arreglemoslo

```
def main():
    n = int(input("Whats n? "))
    for s in sheep(n):
        print(s)

def sheep(n):
    flock = []
    for i in range(n):
        flock.append("🐑" * i)
    return flock

if __name__ == "__main__":
    main()
```

Ahora, existe un problema con este codigo. Si ponemos:
python sleep.py 10, no ocurre nada malo, solo aparecen 10 ovejas.
Pero si ponemos python sleep.py 1000 ya se empieza a deformar el output.
Si pongo 1 000 000, el programa deja de printear sheep porque nos quedamos sin memoria. Para esto tenemos los **generators**.
-> Podemos definir una funcion como generator. donde todavia puede generar una cantidad masiva de data para el user pero va a ir retornando un poquito de esa data cada vez, no todo de una como haciamos antes.
Para eso usaremos **yield** en vez de **return. Yield** e dira a python que retorne una value cada vez que loopeemos.

```
def main():
    n = int(input("Whats n? "))
    for s in sheep(n):
        print(s)

def sheep(n):
    for i in range(n):
        yield "🐑" * i
```

```
if __name__ == “__main__”:
        main()
```

si corro con 1000000 sheeps, funciona, solo que va printeando de a poco, no todo de una como con return. Por eso no se rompe.

Yield retorna un iterador que permite a mi codigo iterar sobre estos generated values uno a la vez