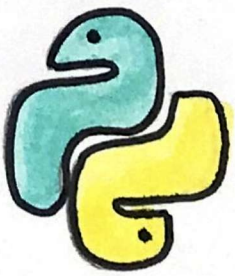
@majoledes

¿Qué es Python?

Es un lenguaje de programación con una sintáxis limpia.

características

- Lenguaje interpretado
- Tipado dinámico
- Fuertemente tipado
- Multiplataforma
- Orientado a objetos



indentación

Python utiliza la indentación para indicar donde empieza y donde termina un bloque de código.

Esto genera un código limpio y fácil de leer ☺

```
def mi_funcion():
    a = 2          ? bloque que pertenece
    return a       ∫ a la función.
print(mi_funcion)
```

comentarios

De una línea

```
#comentario
código
```

De media línea

```
código #comentario
```

De múltiple línea

```
""" comentario
:
"""
```

@majoledes

Variables

Nos permiten guardar valores, ~~permitiéndonos~~ ↳ pudiendo reutilizarlos en diferentes partes del código.

Para crear una variable en Python debes especificar el nombre de la variable y luego asignarle un valor.

```
<variable name> = <valor>
```

↓

OPERADOR DE ASIGNACIÓN

```
a = 10  # Int
x = 'A'  # String
nombre = 'Marcos' #String
```

✦ **Reglas** para nombrar las variables:

- 1. Los nombres de las variables pueden comenzar con una letra o un underscore:

```
x = True
_y = True
```

- 2. Pueden contener letras, números y underscores:

```
variable_1 = 9
```

- 3. Los nombres son case sensitive, es decir que distingue mayúsculas y minúsculas:

```
x = 9
y = X + 5
```

→ NameError: name 'X' is not defined

- 4. Los nombres no deben ser palabras reservadas del lenguaje. (False, class, def, for, break ...)

tipos de datos

booleanos

<class 'bool'>

valor: True o False

números

<class 'int'>

Número sin decimales

a = 2

b = 100

<class 'float'>

Número con decimales

a = 2.0

b = 100.0

<class 'complex'>

Número complejo

a = 2 + 1j

b = 100 - 10j

cadena de texto

<class 'str'>

'esto es una cadena'

las comillas pueden ser simples (' ') ó dobles (" ")

listas

<class 'list'>

Es una colección de datos.

```
a=[1, "lista", 2, 3]
```

Las listas son mutables y se accede a ellas mediante un index.
El primer elemento de una lista está en el indice 0.

tuplas

<class 'tuple'>

Es también una colección de datos pero inmutable.

```
a=(1, 2, 3)
b=('a', 1, 'python')
```

se acceden mediante un index.

diccionarios

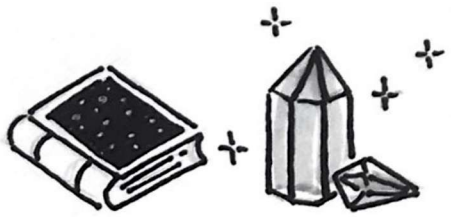
<class 'dict'>

Es una colección desordenada de pares clave-valor.

```
a = { 1 : 'uno',
      2 : 'dos' }
```

Las claves son inmutables y los valores, mutables.
No importa el orden de sus datos.

set



<class 'set'>

colección desordenada de valores únicos.

```
a = {1, 2, 3, 'a', 'c'}
```

set(mi_lista) → quita los elementos repetidos de una lista y el orden que tenía.

mutables e inmutables

Un objeto es llamado mutable si se puede modificar.

Mutables: listas, sets, diccionarios ...

Inmutables: int, float, complex, string, tuplas ...

@majoledes

cadenas de caracteres

Además de números, Python puede manipular cadenas de texto, las cuales pueden estar encerradas en comillas simples ('...') o dobles ("...") con el mismo resultado.

```
>>> 'esto es una cadena'  # comillas simples
'esto es una cadena'

>>> "esto es una cadena"
"esto es una cadena"  # comillas dobles
```

Podemos utilizar # para crear comentarios.

Puedes utilizar el carácter de escape \ para poner comillas del mismo tipo:

```
>>> 'esta \'palabra\' está entre comillas'
esta 'palabra' está entre comillas

>>> "esta \"palabra\" está entre comillas"
esta "palabra" está entre comillas
```

- Las cadenas son inmutables

Operadores matemáticos

- + suma

```
>>> 2 + 2
4
```

- - resta

```
>>> 50 - 10
40
```

- / división: siempre retorna un punto flotante

```
>>> 17 / 3
5.666666666666667 (float)
```

- // división de entero: descarta la parte fraccional

```
>>> 17 // 3
5 (int)
```

- % módulo: retorna el resto de la división

```
>>> 17 % 3
2
```

- ** potencia

```
>>> 5 ** 2
25
```

Operadores lógicos

devuelven un valor booleano

and

si ambos son True

```
x = True
y = True
z = x and y  # z = True
```

or

si al menos uno es True

```
x = True
y = True
z = y or x  # Z = True
```

```
x = True
y = False
z = x or y  # Z = True
```

not

cambia el valor de verdad

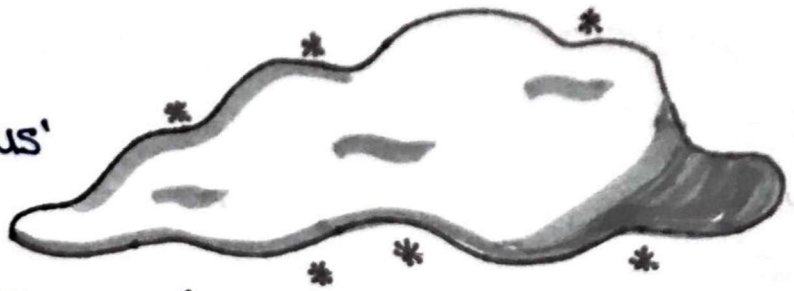
```
x = True
y = not x  # y = False
```

```
x = False
y = not x  # y = True
```

Operaciones con cadenas

- Concatenación (+)

```
>>> 'cumulo' + 'nimbus'
'cumulonimbus'
```



Dos o más cadenas literales (aquellas encerradas entre comillas) una al lado son automáticamente concatenadas:

```
>>> 'Py' 'thon'
'Python'
```

- Multiplicación (*)

```
>>> 3 * 'python'
'pythonpythonpython'
```

Podemos mezclar los operadores:

```
>>> ('Hola' * 3) + ' ' + 'chau'
'HolaHolaHola chau'
```

⚠ Los operadores funcionan según el tipo de dato (type())

Por ejemplo:

```
>>> 5 / 'Centauri'
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

(int, str)

type()

La función type() devuelve el tipo de objeto que recibe como argumento.

type(argumento)

```
>>> my_var = 'Hola Mundo!'
>>> type(my_var)
<class 'str'>
>>> print(my_var)
Hola Mundo!
```

→ STRING

```
>>> my_var = 3
>>> type(my_var)
<class 'int'>
```

→ ENTERO

```
>>> my_var = 4.0
>>> type(my_var)
<class 'float'>
```

→ FLOTANTE

```
>>> type(False)
<class 'bool'>
```

→ BOOLEANO

```
>>> type(4===5)
<class 'bool'>
```

función print()

Es una instrucción que nos permite una salida más legible, omitiendo comillas e imprimiendo caracteres especiales como las tabulaciones \t o los saltos de línea \n

```
>>> print('Esto es una cadena')
Esto es una cadena
>>> print("Hola \t chau")
Hola    chau
>>> print('primera línea.\n segunda línea.')
primera línea.
segunda línea.
```

Si no querés que los caracteres antepuestos por \ sean interpretados, podés utilizar cadenas crudas agregando una r:

```
>>> print(r"c:\nombre\repositorio")  # r(cadena)
c:\nombre\repositorio
```

CONDICIONALES

if, elif, else

En Python puedes definir una serie de condicionales utilizando:

- **if**: primera condición
- **elif**: resto de las condiciones. Puede haber múltiples elif.
- **else**: se ejecuta cuando las anteriores condiciones son falsas.

- `if condición:`
  `bloque de código` } condición = True
- `elif condición2:`
  `bloque de código` } condición = False, condición2 = True
- `else:`
  `bloque de código` } condición = False, condición2 = False

Operador ternario: El orden es diferente a otros lenguajes:

`"mayor que 2" if n>2 else "menor o igual a 2"`

"mayor que 2": bloque a ejecutar; if n>2: condición

iteraciones

nos permiten realizar la misma tarea en varias ocasiones.

while

se repite un bloque mientras la condición lógica devuelva True.

```
contador = 0
while contador < 5:
    print (contador)
    contador +=1 #contador = contador + 1
```

~else~

SE EJECUTA un bloque DE CÓDIGO cuando la condición ya no devuelve TRUE.

```
contador = 0
while contador < 5:
    print (contador)
    contador +=1
else:
    print("Dejo de contar")
```

0
1
2
3
4

"Dejo de contar"

~break~

"Rompe" la ejecución del while en cualquier momento

```
contador = 0
while contador < 5:
    contador +=1
    IF (contador == 4):
        print ("Se rompió")
        break
print (contador)
```

1
2
3

"Se rompió"

~continue~

"Salta" la iteración actual sin romper el bucle

```
contador = 0
while contador < 5:
    contador +=1
    IF (contador == 3 ):
        continue
    print (contador)
```

1
2
4

For

En Python, el bucle for es un "bucle definido", es decir, que preestablece las condiciones de la iteración por adelantado.

```
For <variables> in <iterable>:
    bloque de código
```

```
Frutas = ['kiwi', 'mango', 'cereza']
for Fruta in Frutas:
    print(Fruta)
```

\# Kiwi
mango
cereza

- **iterable**: tipo de dato que se puede secuenciar. Por ejemplo: string, lista, tupla, diccionarios, etc.
- **iterator**: Objeto específico que se obtiene a partir de ese iterable.

```
>>> Frutas=['kiwi','mango','cereza']
>>> iterator = iter(frutas)
>>> next (iterator)
kiwi
>>> next (iterator)
mango
>>> next (iterator)
cereza
```

→ Python llama internamente a la función iter para obtener un iterator

→ Luego, llama repetidamente a la función next para acceder al siguiente elemento del bucle.

- El bucle se detiene una vez que arroja **StopIteration**.

¡ Podemos modificar el comportamiento de un bucle for mediante los keywords break y continue.

Funciones

En Python la definición de funciones se realiza mediante la instrucción def más un nombre de función descriptivo, seguido de paréntesis de apertura y cierre. La definición de la función finaliza con dos puntos (:)

```
def <nombre> (parámetros) :
    <cuerpo>
```

- Siempre tengan en cuenta la **identación** dentro de la función.

```
def saludo ( ) :
    print ("Hola")
```

Para llamar o activar esta función ponemos el nombre de la función seguido de paréntesis.

```
saludo()
```

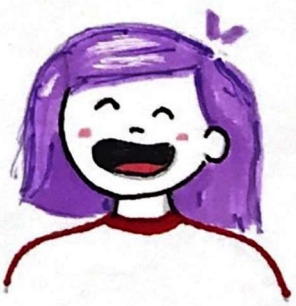
- Con parámetros.

```
def sumar (num1, num2) :
    print (num1 + num2)

sumar (10, 30)   # 40
```

↓

con # hacemos un comentario en Python



función lambda

una función lambda es una función anónima.
Para construir una función lambda, lo hacemos utilizando la palabra lambda. Sus parámetros deben estar separados por comas (sin paréntesis), dos puntos y el código de la función.

```
lambda argumento_uno, argumento_dos : expresión
```

generadores

un generador es un tipo de función que produce secuencias completas de resultados en lugar de ofrecer un único valor.
Devuelve los valores con la declaración yield.

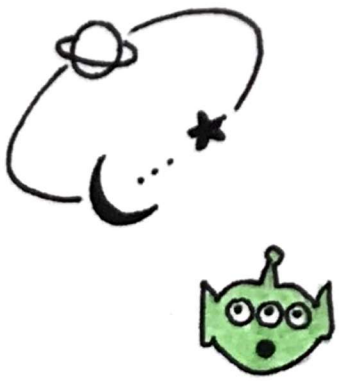
```
def gen_basico():
    yield "uno"
    yield "dos"
    yield "tres"
for valor in gen_basico():
    print(valor) # uno, dos, tres
```

decoradores

Un decorador es un tipo de función la cual recibe como parámetro una función y a su vez retorna otra función.

$$a(b) \rightarrow c$$

```
def funcion_a(funcion_b):
    def funcion_c():
        bloque de código
        funcion_b()
    return funcion_c
```



con el decorador creado, resta decorar una función. Para decorar una función basta con colocar el decorador con el prefijo @ en la parte superior de la función.

```
@funcion_a
def saludo():
    print('Hola ahí!')
```

* Si la función a decorar debe recibir argumentos y, a su vez, retornar algún valor, se utilizan los parámetros args y kwargs los cuales nos permitirán reutilizar el decorador.

al decorar una función estamos modificando su comportamiento sin tener que modificar su código. Esto es útil si queremos crear nuevas funcionalidades.

alcance de las funciones

Las variables que se utilizan deben encontrarse dentro del contexto de ejecución para poder acceder a ellas.

```
variable = 60          → variable global

def funcion():
    variable = 30      → variable local
    IF variable < 100:
        print(variable)

print(variable)        #60
funcion()              #30
print(variable)        #60
```

Utilizando la palabra reservada global nombre_variable es posible crear una variable global dentro de una función

recursividad

Técnica mediante la cual una función se llama a sí misma.

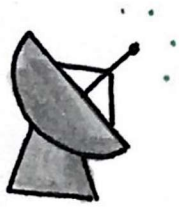
FUNCIÓN RECURSIVA

Python permite a una función llamarse a sí misma de igual forma que lo hace cuando llama a otra función.

$n! = n \cdot (n-1)!$ } Factorial

```
def factorial(n):
    """Calcula el factorial de n
    n int>0
    return n!
    """
#comenzamos con el caso base
    IF n == 1:
        return 1
#definición matemática
    return n * factorial(n-1)
n = int(input('Escribe un entero: '))
print(factorial(n))
```

ESPECIFICACIÓN



La función factorial se invoca recursivamente...

Manejo de excepciones .-

EXCEPCIONES: Errores detectados durante la ejecución.

¿Cómo las manejamos ?

Con 3 keywords: TRY, EXCEPT, FINALLY

```
>>> while True:
        try: # intenta ejecutar el código
            x = int(input("Por favor ingrese un número: "))
            break
        except ValueError: # maneja el error
            print("Oops! No era válido. Intenta de nuevo...")
```

Finally: Una cláusula finally siempre es ejecutada antes de salir de la declaración try, ya sea que una excepción haya ocurrido o no.

```
>>> try:
        raise KeyboardInterrupt
    Finally:
        print("Chau, Mundo")
...
Chau, Mundo
KeyboardInterrupt
```

¿Cómo funciona try?

- ejecuta el bloque `try` (entre try y except)
- si no ocurre ninguna excepción, el bloque `except` se saltea y termina la ejecución de la declaración `try`.
- si ocurre una excepción dentro del bloque `try`, el resto del bloque se saltea. Si el tipo de Excepción corresponde con la nombrada luego de `except`, se ejecuta el bloque `except`, y la ejecución continúa luego de la declaración `try`.
- la declaración `try` puede tener más de un `except` para manejar distintas excepciones.
- las declaraciones `try`...`except` tienen un bloque `else` opcional que continúa al `except`.
  Es útil para aquel código que debe ejecutarse si el bloque `try` no genera una excepción.

la declaración `raise` permite forzar a que ocurra una excepción específica.

```
raise NombreDelaExcepción
```

CLASES y OBJETOS

Casi todo en Python es un objeto, con sus propiedades y métodos.

Una clase es como un constructor de objetos.

Para crear una clase, utilizamos la palabra class.

Todas las clases tienen un método llamado __init__() que siempre se ejecuta cuando se inicia la clase, y es usado para asignar valores a las propiedades del objeto u otras operaciones necesarias.

El parámetro self es una referencia al objeto actual, y se utiliza para acceder a los atributos y métodos del objeto.

```
class Monster:
  def __init__(self, nombre, categoría):
    self.nombre = nombre
    self.categoría = categoria

  def myFunc(self)
    print("Hey, yo soy " + self.nombre)

#Instancio la clase. creo el objeto.
monstruito = Monster("Sullivan", "asustador")

monstruito.myFunc()
```

Método: es una función dentro de nuestra clase

Método: acción que realiza el objeto

@majodedes