Notas de Aprendizaje del Tutorial de Python

A través del aprendizaje previo, ya hemos adquirido algo de conocimiento sobre Python. Ahora, siguiendo la documentación oficial, continuaremos complementando nuestro conocimiento sobre Python.

Control del flujo de código

tipo

```
print(type(1))
```

```
<class 'int'>
```

```
print(type('a'))
```

```
<class 'str'>
```

La función `type` es muy útil para imprimir el tipo de un objeto.

range

El término **range** en programación se refiere a una secuencia de números generados dentro de un intervalo específico. Es comúnmente utilizado en bucles para iterar sobre una serie de valores. A continuación, te muestro algunos ejemplos de cómo se utiliza en diferentes lenguajes de programación:

En Python:

```python
# Generar un rango de 0 a 9
for i in range(10):
    print(i)
```

En JavaScript:

```javascript
// Generar un rango de 0 a 9
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

En Ruby:

```ruby
# Generar un rango de 0 a 9
(0..9).each do |i|
  puts i
end
```

El uso de `range` puede variar ligeramente dependiendo del lenguaje, pero el concepto general es el mismo: generar una secuencia de números dentro de un intervalo definido.

La función `range` es extremadamente útil.

```python
for i in range(5):
  print(i, end = ' ')
```

Nota: El código no necesita traducción, ya que es un bloque de código en Python y los comandos y sintaxis son universales en el lenguaje de programación.

```
0 1 2 3 4
```

```python
for i in range(2, 6, 2):
  print(i, end = ' ')
```

El código anterior en Python utiliza un bucle `for` para iterar sobre un rango de números. Aquí está la traducción al español de lo que hace:

```python
for i in range(2, 6, 2):
  print(i, end = ' ')
```

Este código en Python utiliza un bucle `for` para iterar sobre un rango de números que comienza en 2, termina en 6 (sin incluir el 6) y avanza de 2 en 2. Luego, imprime cada valor de `i` seguido de un espacio en lugar de un salto de línea.

```
2 4
```

Observa la definición de la función `range`.

```
class range(Sequence[int]):
    start: int
    stop: int
    step: int
```

Nota: El código anterior está en Python y no necesita traducción, ya que es un bloque de código que define una clase `range` con atributos `start`, `stop` y `step`.

`Visible` es una clase.

```
print(range(5))
```

```
range(0, 5)
```

En lugar de:

```
[0,1,2,3,4]
```

Continúa.

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

¿Por qué? Mira la definición de `list`.

```
class list(MutableSequence[_T], Generic[_T]):
```

En este fragmento de código, se define una clase `list` que hereda de `MutableSequence` y es genérica con respecto al tipo `_T`. Esto indica que la lista puede contener elementos de cualquier tipo `_T`.

La definición de `list` es `list(MutableSequence[_T], Generic[_T]):`. Mientras que la definición de `range` es `class range(Sequence[int])`. `list` hereda de `MutableSequence`. `range` hereda de `Sequence`.

Continuando hacia abajo, se encuentra lo siguiente.

```
Sequence = _alias(collections.abc.Sequence, 1)
MutableSequence = _alias(collections.abc.MutableSequence, 1)
```

Aquí no entendemos la relación entre ellos. Pero probablemente ya sabemos por qué podemos escribir `list(range(5))`.

Parámetros de Funciones

Vamos a ver conocimientos adicionales sobre funciones.

```
def fn(a = 3):
  print(a)
```

fn()

```shell
3
```

Esto es para asignar un valor predeterminado a un parámetro.

```
def fn(end: int, start = 1):
  i = start
  s = 0
  while i < end:
    s += i
    i += 1
  return s
```

El código anterior define una función llamada `fn` que toma dos argumentos: `end` (un entero que indica el límite superior) y `start` (un entero opcional que indica el valor inicial, por defecto es 1). La función suma todos los números enteros desde `start` hasta `end - 1` y devuelve el resultado de la suma.

```
print(fn(10))
```

```
45
```

El parámetro `end` es obligatorio. Ten en cuenta que los parámetros obligatorios deben escribirse al principio.

```
def fn(start = 1, end: int):
```

En este fragmento de código, se define una función llamada `fn` que tiene dos parámetros:

- `start`: Un parámetro con un valor predeterminado de `1`.

- `end`: Un parámetro que espera un valor de tipo `int` (entero).

Nota: En Python, los tipos de los parámetros se pueden anotar utilizando la sintaxis de anotaciones de tipo, como se muestra en el parámetro `end`. Sin embargo, el parámetro `start` no tiene una anotación de tipo explícita.

```
    def fn(start = 1, end: int):
                                ^
SyntaxError: argumento no predeterminado sigue a un argumento predeterminado
```

Observa que `end` es un `non-default argument` (argumento no predeterminado). `start` es un `default argument` (argumento predeterminado). Esto significa que un argumento no predeterminado sigue a un argumento predeterminado. Es decir, los argumentos no predeterminados deben colocarse antes de todos los argumentos predeterminados. `start` es un argumento predeterminado, lo que significa que si no se pasa, ya tiene un valor por defecto.

```
def fn(a, /, b):
  print(a + b)
```

En este código, la función `fn` toma dos parámetros: `a` y `b`. El parámetro `a` es un parámetro posicional solamente (indicado por la barra `/`), lo que significa que no puede ser pasado como un argumento de palabra clave. El parámetro `b` puede ser pasado tanto posicionalmente como por palabra clave. La función simplemente imprime la suma de `a` y `b`.

fn(1, 3)

```
Aquí se utiliza `/` para separar los tipos de parámetros. Hay dos formas de pasar parámetros: una es po

```python
def fn(a, /, b):
  print(a + b)
```

fn(a=1, 3)

```shell
    fn(a=1, 3)
               ^
SyntaxError: argumento posicional sigue a un argumento de palabra clave
```
```

Así no funcionará. `a=1` indica que se está pasando un argumento por palabra clave. Esto lo convierte en un argumento de palabra clave, mientras que `b` es un argumento posicional.

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------    ----------     ----------
        |             |                  |
        |        Posicional o por palabra clave |
        |                                - Solo por palabra clave
         -- Solo posicional
```

Ten en cuenta que al definir la función aquí, el uso de `/` y `*` ya implica el tipo de paso de los parámetros. Por lo tanto, debes seguir las reglas al pasarlos.

```
def fn(a, /, b):
  print(a + b)
```

En este código, la función `fn` toma dos argumentos: `a` y `b`. El uso de `/` en la definición de la función indica que `a` es un parámetro posicional, lo que significa que no puede ser pasado como un argumento de palabra clave. Por otro lado, `b` puede ser pasado tanto como un argumento posicional como de palabra clave. La función simplemente imprime la suma de `a` y `b`.

fn(1, b=3)

```
Así no hubo errores.

```python
def fn(a, /, b, *, c):
  print(a + b + c)
```

fn(1, 3, 4)

```shell
    fn(1, 3, 4)
TypeError: fn() toma 2 argumentos posicionales pero se proporcionaron 3
```

`fn` solo puede recibir 2 argumentos posicionales, pero se proporcionaron 3.

```
def fn(a, /, b, *, c):
  print(a + b + c)
```
```

En este código, la función `fn` tiene tres parámetros: `a`, `b` y `c`. La barra diagonal `/` antes de `b` indica que `a` es un parámetro posicional solamente, lo que significa que no puede ser pasado como un argumento de palabra clave. El asterisco `*` antes de `c` indica que `c` debe ser pasado como un argumento de palabra clave, no posicional. La función simplemente imprime la suma de `a`, `b` y `c`.

fn(a = 1, b=3, c=4)

```shell
    fn(a = 1, b=3, c=4)
TypeError: fn() recibió algunos argumentos posicionales pasados como argumentos de palabra clave: 'a'
```

En `fn`, algunos parámetros que antes solo se podían pasar por posición ahora se pueden pasar usando palabras clave.

Parámetros en Forma de Mapeo

```
def fn(**kwds):
  print(kwds)
```

fn(**{‘a’: 1})

```shell
{'a': 1}
```

```
def fn(**kwds):
  print(kwds['a'])

d = {'a': 1}
fn(**d)
```

```
1
```

El uso de `**` permite expandir los parámetros.

```
def fn(a, **kwds):
  print(kwds['a'])
```

```
d = {'a': 1}
fn(1, **d)

TypeError: fn() recibió múltiples valores para el argumento 'a'
```

Cuando llamas a una función como `fn(1, **d)`, se expande a `fn(a=1, a=1)`. Por lo tanto, ocurrirá un error.

```
def fn(**kwds):
  print(kwds['a'])
```

d = {‘a’: 1} fn(d)

```shell
TypeError: fn() toma 0 argumentos posicionales pero se proporcionó 1
```

Si llamas a una función como `fn(d)`, se tratará como un argumento posicional, no se expandirá como un argumento de palabra clave.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

En este código, la función `fn` toma un argumento posicional `a` y un conjunto de argumentos de palabras clave `**kwds`. La barra `/` indica que todos los parámetros antes de ella deben ser pasados como argumentos posicionales, no como palabras clave. Luego, la función imprime el valor asociado con la clave `'a'` en el diccionario `kwds`.

```
d = {'a': 1}
fn(1, **d)
```

Esto funciona. Indica que los parámetros posicionales y los parámetros en forma de mapeo pueden tener el mismo nombre.

```
def fn(a, / , a):
  print(a)
```

En este código, la función `fn` tiene dos parámetros llamados `a`. El uso de `/` en la definición de la función indica que todos los parámetros antes de `/` deben ser pasados de manera posicional y no pueden ser usados como argumentos de palabra clave. Sin embargo, tener dos parámetros con el mismo nombre `a` causará un error de sintaxis en Python, ya que los nombres de los parámetros deben ser únicos.

```
d = {'a': 1}
fn(1, **d)

SyntaxError: argumento duplicado 'a' en la definición de la función
```

Así es como ocurre un error. Presta atención a las sutiles relaciones entre estas situaciones.

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

En este código, la función `fn` toma un argumento posicional `a` y un diccionario de argumentos de palabras clave `**kwds`. La barra `/` en la definición de la función indica que todos los parámetros antes de la barra deben ser pasados como argumentos posicionales, no como palabras clave. Luego, la función imprime el valor asociado con la clave `'a'` en el diccionario `kwds`.

fn(1, **[1,2])

```
En este caso, el código no necesita ser traducido, ya que es una sintaxis específica de programación. S

La función `fn` está siendo llamada con dos argumentos: el número `1` y una lista `[1, 2]` que está sie

```shell
TypeError: el argumento después de ** en __main__.fn() debe ser un mapeo, no una lista
```

`**` debe ir seguido de un mapeo.

Parámetros de tipos iterables

```
def fn(*kwds):
  print(kwds)
```

fn(*[1,2])

```
En este caso, el código no necesita ser traducido, ya que es una expresión de Python que utiliza la sin

```shell
(1, 2)
```

```
def fn(*kwds):
  print(kwds)
```

fn(*1)

```shell
TypeError: el argumento después de * en __main__.fn() debe ser un iterable, no un int
```

`*` debe seguir a `iterable`.

```
def fn(a, *kwds):
  print(type(kwds))
```

fn(1, *[1])

```shell
<class 'tuple'>
```

Imprime el tipo. Esto también explica por qué la salida anterior es `(1,2)` en lugar de `[1,2]`.

```
def fn(*kwds):
  print(kwds)
```

fn(1, *[1])

```shell
(1, 1)
```

Observa que cuando se llama a `fn(1, *[1])`, los parámetros se descomprimen, convirtiéndose en `fn(1, 1)`. Luego, cuando se analiza `fn(*kwds)`, `kwds` convierte `1, 1` en la tupla `(1, 1)`.

```
def concat(*args, sep='/'):
  return sep.join(args)

print(concat('a','b','c', sep=','))
```

```
a,b,c
```

Expresiones Lambda

`lambda` es una forma de guardar una función como si fuera una variable. ¿Recuerdas lo que mencionamos en el artículo “Desmitificando la Ciencia de la Computación”?

```
def incrementor(n):
  return lambda x: x + n

f = incrementor(2)
print(f(3))
```

```
5
```

Nota: El número “5” no necesita traducción, ya que es un valor numérico y se mantiene igual en ambos idiomas.

Veamos otro ejemplo.

```
pares = [(1, 4), (2, 1), (0, 3)]
```

pairs.sort(key = lambda par: par[1])

```
print(pairs)
```

```
[(2, 1), (0, 3), (1, 4)]
```

```
pares = [(1, 4), (2, 1), (0, 3)]
```

pairs.sort(key = lambda par: par[0])

```
print(pairs)
```

```
[(0, 3), (1, 4), (2, 1)]
```

Cuando es `pair[0]`, se ordena según el primer número. Cuando es `pair[1]`, se ordena según el segundo número.

Comentarios de Documentación

```
def add():
  """añadir algo
  """
  pass

print(add.__doc__)
```

```
agregar algo
```

Firma de la Función

```
def add(a:int, b:int) -> int:
  print(add.__annotations__)
  return a+b

add(1, 2)
```

```
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Estructuras de Datos

Listas

```
a = [1,2,3,4]
```

a.append(5) print(a) # [1, 2, 3, 4, 5]

a[len(a):] = [6] print(a) # [1, 2, 3, 4, 5, 6]

a[3:] = [6] print(a) # [1, 2, 3, 6]

```
a.insert(0, -1)
print(a)   # [-1, 1, 2, 3, 6]
```

a.remove(1) print(a) # [-1, 2, 3, 6]

a.pop() print(a) # [-1, 2, 3]

```
a.clear()
print(a)  # []
```

a[:] = [1, 2] print(a.count(1)) # 1

a.reverse() print(a) # [2, 1]

b = a.copy() a[0] = 10 print(b) # [2, 1] print(a) # [10, 1]

```
b = a
a[0] = 3
print(b)  # [3, 1]
print(a)  # [3, 1]
```

Construcción de Listas

```
print(3 ** 2)   # 9
print(3 ** 3)   # 27
```

Primero, aprendamos una operación: `**`. Esto significa `elevado a la potencia de`.

```
sq = []
for x in range(10):
  sq.append(x ** 2)

print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Luego, prueba usando `map`.

```
a = map(lambda x:x, range(10))
print(a)
# <map object at 0x103bb0550>
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sq = map(lambda x: x ** 2, range(10))
print(list(sq))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
sq = [x ** 2 for x in range(10)]
print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Se puede ver que `for` es muy flexible.

```
a = [i for i in range(5)]
print(a)
# [0, 1, 2, 3, 4]
```

a = [i+j for i in range(3) for j in range(3)] print(a) # [0, 1, 2, 1, 2, 3, 2, 3, 4]

a = [i for i in range(5) if i % 2 == 0] print(a) # [0, 2, 4]

```
a = [(i,i) for i in range(3)]
print(a)
# [(0, 0), (1, 1), (2, 2)]
```

Construcción de Listas Anidadas

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

```
t = []
for j in range(3):
  t.append([(i+j*4) for i in range(4)])
print(t)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

Fíjate en la forma de estos dos fragmentos de código. Es decir:

```
[[(i+j*4) for i in range(4)] for j in range(3)]
```

El código anterior genera una lista de listas en Python. No es necesario traducir el código, ya que es universal y funciona de la misma manera en cualquier idioma. Sin embargo, si deseas una explicación en español:

Este código utiliza una comprensión de lista anidada para crear una matriz de 3 filas y 4 columnas. La expresión `(i + j * 4)` calcula el valor de cada elemento en la matriz, donde `i` es el índice de la columna y `j` es el índice de la fila. El resultado será:

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

Cada sublista representa una fila de la matriz.

Lo que equivale a:

```
for j in range(3):
   [(i+j*4) for i in range(4)]
```

Es decir, es equivalente a:

```
for j in range(3):
  for i in range(4):
      (i+j*4)
```

El código anterior no tiene una salida visible, ya que no se está imprimiendo ni almacenando el resultado de la expresión `(i + j * 4)`. Si quisieras ver los valores generados, podrías modificar el código para imprimirlos, por ejemplo:

```
for j in range(3):
  for i in range(4):
      print(i + j * 4)
```

Esto generaría la siguiente salida:

```
0
1
2
3
4
5
6
7
8
9
10
11
```

Por lo tanto, esto es conveniente para realizar la transposición de matrices.

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
#  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

mt = [[row[j] for row in matrix] for j in range(4)] print(mt) # [[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]

```
print(list(zip(*matrix)))
[(0, 4, 8), (1, 5, 9), (2, 6, 10), (3, 7, 11)]
```

del

El operador `del` en Python se utiliza para eliminar objetos. Puedes usarlo para eliminar variables, elementos de una lista, claves de un diccionario, y más. Aquí tienes algunos ejemplos:

Eliminar una variable

```
x = 10
del x
# Ahora x ya no está definida
```

Eliminar un elemento de una lista

```
lista = [1, 2, 3, 4, 5]
del lista[2]  # Elimina el elemento en la posición 2 (el número 3)
print(lista)  # Salida: [1, 2, 4, 5]
```

Eliminar una clave de un diccionario

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
del diccionario['b']  # Elimina la clave 'b'
print(diccionario)  # Salida: {'a': 1, 'c': 3}
```

Eliminar un segmento de una lista

```
lista = [1, 2, 3, 4, 5]
del lista[1:3]  # Elimina los elementos desde la posición 1 hasta la 2
print(lista)  # Salida: [1, 4, 5]
```

El operador `del` es una herramienta poderosa, pero debes usarlo con cuidado, ya que elimina objetos de manera permanente.

```
a = [1, 2, 3, 4]

del a[1]
print(a)  # [1, 3, 4]
```

del a[0:2] print(a) # [4]

```
del a
print(a) # NameError: name 'a' is not defined
```

Diccionarios

```
ages = {'li': 19, 'wang': 28, 'he' : 7}
for name, age in ages.items():
    print(name, age)
```

li 19

wang 28

he 7

```
for name in ages:
    print(name)

# li
# wang
# he
```

```
for name, age in ages:
     print(name)
```

ValueError: demasiados valores para desempaquetar (se esperaban 2)

```
for i, name in enumerate(['li', 'wang', 'he']):
    print(i, name)
```

0 li

1 wang

2 he

print(reversed([1, 2, 3])) # <list_reverseiterator object at 0x10701ffd0>

```
print(list(reversed([1, 2, 3])))
# [3, 2, 1]
```

Módulos

Llamar a un módulo mediante un script

```python
import sys

def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

if __name__ == "__main__":
    r = f(int(sys.argv[1]))
    print(r)
```

```
% python fib.py 3
2

% python -m fib 5
5
```

dir

```
import fib

print(dir(fib))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__
```

```
import builtins
print(dir(builtins))
```

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '**build_class**', '**debug**', '**doc**', '**import**', '**loader**', '**name**', '**package**', '**spec**', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

Paquete

Paquetes, es decir, `packages`.

```shell
pk.py
fibp
   cal
      cal.py
   pt
       pt.py
```

cal.py:

```
def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

def fl(n):
    return list(map(f, range(5)))
```

pt.py:

```
def p(l):
    print(l, end=' ')

def pln(l):
    print(l)
```

pk.py:

```
import fibp.cal.cal
import fibp.pt.pt
```

fibp.pt.pt.p(fibp.cal.cal.fl(10))

`pk.py` también se puede escribir de la siguiente manera:

```python
from fibp.cal import cal
from fibp.pt import pt
```

pt.p(cal.fl(10)) “‘