Qué función colocaría en la línea 1 para que el resultado que imprime la línea 2 sea <class 'str'>

1. x = 5

2. print(type(x))

Puede obtener el tipo de datos de cualquier objeto utilizando la función type():

print(type(x))

Si desea especificar el tipo de datos, puede utilizar las siguientes funciones de construcción:

|  |  |
| --- | --- |
| **Example** | **Data Type** |
| x = str("Hello World") | str |
| x = int(20) | int |
| x = float(20.5) | float |
| x = complex(1j) | complex |
| x = list(("apple", "banana", "cherry")) | list |
| x = tuple(("apple", "banana", "cherry")) | Tuple |
| x = range(6) | Range |
| x = dict(name="John", age=36) | Dict |
| x = set(("apple", "banana", "cherry")) | Set |
| x = frozenset(("apple", "banana", "cherry")) | Frozenset |
| x = bool(5) | Bool |
| x = bytes(5) | Bytes |
| x = bytearray(5) | Bytearray |
| x = memoryview(bytes(5)) | Memoryview |

**Respuesta:**

x = '5' ó x = str(5)

Mencione los tipos numéricos que posee Python y vistos en clase

Python tiene los siguientes tipos de datos integrados de forma predeterminada, en estas categorías:

|  |  |
| --- | --- |
| Text Type: | str |
| Numeric Types: | int, float, complex |
| Sequence Types: | list, tuple, range |
| Mapping Type: | dict |
| Set Types: | set, frozenset |
| Boolean Type: | bool |
| Binary Types: | bytes, bytearray, memoryview |
| None Type: | NoneType |

**Respuesta:**

int, float, complex

Considerando los tipos de secuencia y si decimos que es una colección ordenada e inmutable nos referimos a:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Lista** | **Tupla** | **Conjunto** | **Diccionario** |
| **Delimitador** | [] | () | {} | {} |
| **¿Es mutable?** | Sí | No | Sí  Sin duplicados | Sí  Sin duplicados |
| **¿Es ordenado?** | Sí | Sí | No | Sí |

**Respuesta:**

Tupla

Qué se observa en la consola si ejecutamos el siguiente código:

numeros = (1, 2, 3)

resultado = numeros \* 2

print(resultado)

Si desea multiplicar el contenido de una tupla un número determinado de veces, puede utilizar el operador \*.

**Ejemplo.** Multiplica la tupla de frutas por 2:

fruits = ("apple", "banana", "cherry")  
mytuple = fruits \* 2  
print(mytuple)

**Respuesta:**

('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')

Considerando los tipos de secuencia y si decimos que es una colección desordenada, inmutable y no indexada nos referimos a:

(Entendemos por inmutable a la no posibilidad de realizar cambios pero sí a eliminar y agregar datos)

**Colecciones de Python (matrices/arrays)**

Hay cuatro tipos de datos de recopilación en el lenguaje de programación Python:

* La **lista** es una colección ordenada y modificable. Permite miembros duplicados.
* La **tupla** es una colección ordenada e inmutable. Permite miembros duplicados.
* El **conjunto** es una colección desordenada, inmutable**\*** y no indexada. No hay miembros duplicados.
* El diccionario es una colección ordenada**\*\*** y modificable. No hay miembros duplicados.

**\***Los elementos establecidos no se pueden cambiar, pero puede eliminar y/o agregar elementos cuando lo desee.

**\*\***A partir de la versión 3.7 de Python, se ordenan los diccionarios. En Python 3.6 y versiones anteriores, los diccionarios están desordenados.

Al elegir un tipo de colección, es útil comprender las propiedades de ese tipo. Elegir el tipo correcto para un conjunto de datos en particular podría significar la retención del significado y podría significar un aumento en la eficiencia o la seguridad.

**Respuesta:**

El conjunto (set)

Indique que colocaría en las líneas punteadas para observar 2007:

child1 = {

"name" : "Emil",

"year" : 2004

}

child2 = {

"name" : "Tobias",

"year" : 2007

}

child3 = {

"name" : "Linus",

"year" : 2011

}

myfamily = {

"child1" : child1,

"child2" : child2,

"child3" : child3

}

print(myfamily.................)

La diferencia en la "apariencia" entre un json y un dictionary:

import json

# Un JSON:

jayson = '{"name": "John", "age": 30, "city": "New York"}'

# Procesar jayson:

diccionario = json.loads(jayson)

# El resultado es un diccionario:

print(diccionario)

# Salida: {'name': 'John', 'age': 30, 'city': 'New York'}

**Respuesta:**

print(myfamily["child2"]["year"])

Complete que pondría en 1,2 y 3 para poder ver números desde el 0 hasta el 5

for ....1....... in range(......2......):

print(.....3.......)

**Funciones integradas de Python**

Python tiene un conjunto de funciones integradas. range() devuelve una secuencia de números, a partir de 0 y se incrementa en 1 (por defecto), y se detiene antes de un número específico.

Sintaxis: rango (inicio, parada, paso)

**Respuesta:**

for x in range(0, 6, 1):

print(x)

Complete que pondría en 1, 2, 3, 4, 5 y 6 para poder ver el doble del valor ingresado en 6

def myfunc(1):

return lambda 2 : 3 \* 4

mydoubler = myfunc(5)

print(mydoubler(6))

**¿Por qué utilizar las funciones de Lambda?**

El poder de lambda se muestra mejor cuando los usa como una función anónima dentro de otra función.

Digamos que tiene una definición de función que toma un argumento, y ese argumento se multiplicará con un número desconocido:

def myfunc(n):  
 return lambda a : a \* n

Use esa definición de función para hacer una función que siempre duplique el número que envía:

**Respuesta:**

def myfunc(n):  
 return lambda a : a \* n

mydoubler = myfunc(2)

print(mydoubler(11))

Complete la clase Person para que posea la función constructora de acuerdo al código que se observa más abajo

class Person:

................................................

...............................................

p1 = Person("John", 36)

print(p1.name)

print(p1.age)

**Respuesta:**

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 p1 = Person("John", 36)

print(p1)

Complete la línea de puntos para que se vea 30:

import json

x = '{ "name":"John", "age":30, "city":"New York"}'

y = json...........................(x)

print(y["age"])

**Respuesta:**

import json

x = '{ "name":"John", "age":30, "city":"New York"}'

y = json.loads(x)

print(y["age"])