

Functions with Python



Defining a Function

UNA FUNCIÓN CONSTA DE MUCHAS PARTES, ASÍ QUE PRIMERO VAMOS A FAMILIARIZARNOS CON SU NÚCLEO: UNA DEFINICIÓN DE FUNCIÓN.

AQUÍ HAY UN EJEMPLO DE UNA DEFINICIÓN DE FUNCIÓN:

```
def function_name():
    # functions tasks go here

def trip_welcome():
    print("Welcome to Tripcademy!")
    print("Let's get you to your destination.")
```

Calling a Function

EL PROCESO DE EJECUTAR EL CÓDIGO DENTRO DEL CUERPO DE UNA FUNCIÓN SE CONOCE COMO LLAMARLO (ESTO TAMBIÉN SE CONOCE COMO "EJECUTAR UNA FUNCIÓN"). PARA LLAMAR A UNA FUNCIÓN EN PYTHON, ESCRIBA SU NOMBRE SEGUIDO DE PARÉNTESIS () .

REVISEMOS NUESTRA FUNCIÓN
DIRECTION_TO_TIMESSQ() :

```
def directions_to_timesSq():
    print("Walk 4 mins to 34th St Herald Square train station.")
    print("Take the Northbound N, Q, R, or W train 1 stop.")
    print("Get off the Times Square 42nd Street stop.")
```

directions_to_timesSq()

Walk 4 mins to 34th St Herald Square train station.
Take the Northbound N, Q, R, or W train 1 stop.
Get off the Times Square 42nd Street stop.

Whitespace & Execution Flow

En Python, la cantidad de espacios en blanco le dice a la computadora qué es parte de una función y qué no es parte de esa función.

Si quisiéramos escribir otra declaración fuera de `trip_welcome()`, tendríamos que eliminar la sangría de la nueva línea:

```
def trip_welcome():
    # Indented code is part of the function body
    print("Welcome to Tripcademy!")
    print("Let's get you to your destination.")

# Unindented code below is not part of the function body
print("Woah, look at the weather outside! Don't walk, take the train!")
```

trip_welcome()

Woah, look at the weather outside! Don't walk, take the train!

Welcome to Tripcademy!

Let's get you to your destination.

Parameters & arguments

- El parámetro es el nombre definido entre paréntesis de la función y se puede utilizar en el cuerpo de la función.
- El argumento son los datos que se pasan cuando llamamos a la función y se asignan al nombre del parámetro.

```
def trip_welcome(destination):  
    print("Welcome to Tripcademy!")  
    print("Looks like you're going to " + destination + " today.")
```

```
trip>Welcome("Times Square")
```

Welcome to Tripcademy!

Looks like you're going to Times Square today.

PARAMETER

```
def trip_welcome(destination):  
    print(" Welcome to Tripcademy! ")  
    print(" Looks like you're going to the " + destination + " today. ")
```

PARAMETERS ARE TREATED LIKE VARIABLES WITHIN A FUNCTION

ARGUMENT AS VALUES

```
trip_welcome("Empire State Building")
```

Multiple Parameters

Podemos escribir una función que tome más de un parámetro usando comas

Cuando llamemos a nuestra función, necesitaremos proporcionar argumentos para cada uno de los parámetros que asignamos en nuestra definición de función.

```
def trip_welcome(origin, destination):  
    print("Welcome to Tripcademy")  
    print("Looks like you are traveling from " + origin)  
    print("And you are heading to " + destination)
```

```
trip_welcome("Prospect Park", "Atlantic Terminal")
```

Welcome to Tripcademy

Looks like you are traveling from Prospect Park

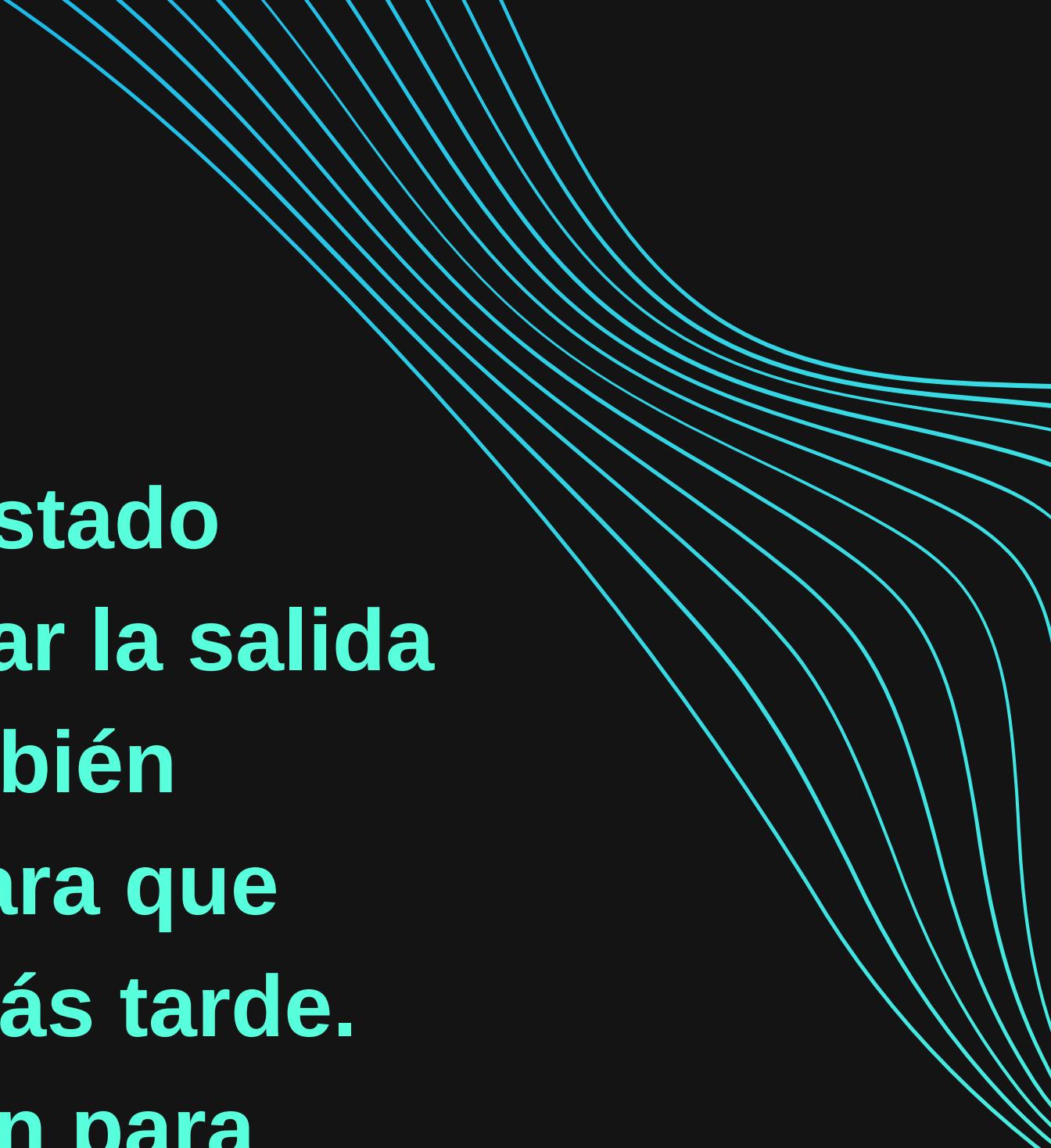
And you are heading to Atlantic Terminal

Types of Arguments

- **Argumentos posicionales:** argumentos que pueden ser llamados por su posición en la definición de la función.
- **Argumentos de palabras clave:** argumentos que se pueden llamar por su nombre.
- **Argumentos predeterminados:** argumentos que reciben valores predeterminados.

```
def calculate_taxi_price(miles_to_travel, rate, discount):  
    print(miles_to_travel * rate - discount )  
  
# 100 is miles_to_travel  
# 10 is rate  
# 5 is discount  
calculate_taxi_price(100, 10, 5)  
  
calculate_taxi_price(rate=0.5, discount=10, miles_to_travel=100)
```

Returns



En este punto, nuestras funciones han estado usando `print()` para ayudarnos a visualizar la salida en nuestro intérprete. Las funciones también pueden devolver un valor al programa para que este valor pueda modificarse o usarse más tarde. Usamos la palabra clave `return` de Python para hacer esto.

```
def calculate_exchange_usd(us_dollars, exchange_rate):  
    return us_dollars * exchange_rate
```

```
new_zealand_exchange = calculate_exchange_usd(100, 1.4)
```

```
print("100 dollars in US currency would give you "  
+ str(new_zealand_exchange) + " New Zealand dollars")
```

100 dollars in US currency would give you 140 New Zealand dollars

Multiple Returns

A veces es posible que queramos devolver más de un valor de una función. Podemos devolver varios valores separándolos con una coma. Eche un vistazo a este ejemplo de una función que permite a los usuarios de nuestra aplicación de viajes comprobar el tiempo de la próxima semana (a partir del lunes):

```
weather_data = ['Sunny', 'Sunny', 'Cloudy', 'Raining', 'Snowing']

def threeday_weather_report(weather):
    first_day = " Tomorrow the weather will be " + weather[0]
    second_day = " The following day it will be " + weather[1]
    third_day = " Two days from now it will be " + weather[2]
    return first_day, second_day, third_day

monday, tuesday, wednesday = threeday_weather_report(weather_data)

print(monday)
print(tuesday)
print(wednesday)
```

Tomorrow the weather will be Sunny

The following day it will be Sunny

Two days from now it will be cloudy

Modules Python Introduction

Un módulo es una colección de declaraciones de Python destinadas en general a ser utilizadas como una herramienta. Los módulos también se denominan a menudo "bibliotecas" o "paquetes": un paquete es realmente un directorio que contiene una colección de módulos.

```
from module_name import object_name
```

Modules Python Random

Con `random`, usaremos más de una pieza de la funcionalidad del módulo, por lo que la sintaxis de importación se verá así:

- `random.choice()`
- `random.randint()`

Modules Python Namespaces

```
import codecademylib3_seaborn

# Add your code below:

from matplotlib import pyplot as plt
import random

numbers_a = range(1, 13)
numbers_b = random.sample(range(1000), 12)
plt.plot(numbers_a, numbers_b)
plt.show()
```

Modules Python Files and Scope

Los archivos son en realidad módulos, por lo que puede dar acceso a un archivo al contenido de otro archivo usando esa import declaración gloriosa.

× script.py

library.py

script.py × library.py

```
# Add your always_three() function below:  
▼ def always_three():  
    return 3
```

```
# Import library below:
```

```
from library import always_three
```

```
# Call your function below:
```

```
always_three()
```

Introduction to Python Dictionaries

Un diccionario es un conjunto desordenado de key:value pares.

Nos proporciona una forma de mapear piezas de datos entre sí para que podamos encontrar rápidamente valores que están asociados entre sí.

- Avocado Toast is 6 dollars
- Carrot Juice is 5 dollars
- Blueberry Muffin is 2 dollars

```
menu = {"avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2}
```

```
subtotal_to_total = {20: 24, 10: 12, 5: 6, 15: 18}
```

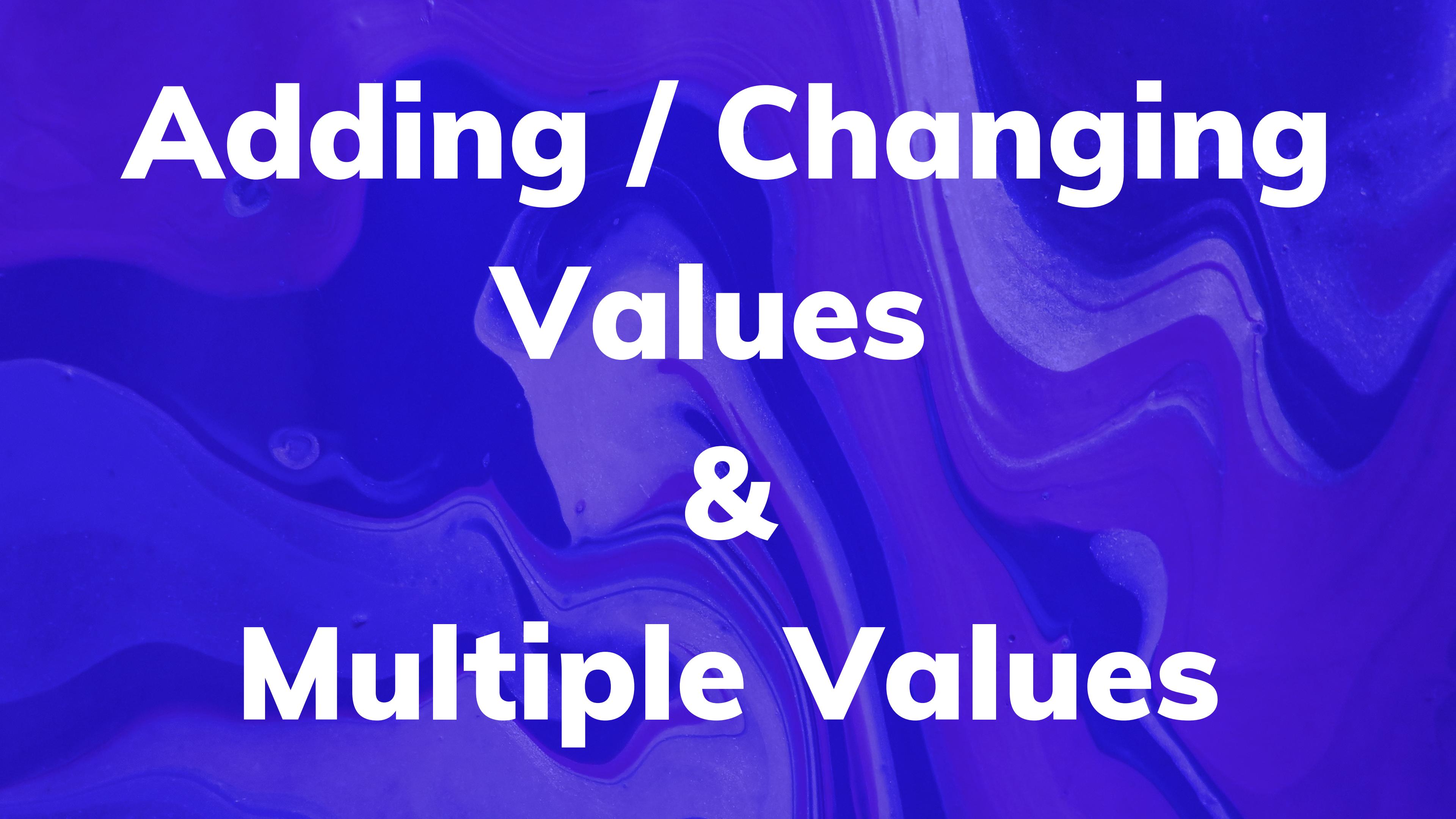
```
students_in_classes = {"software design": ["Aaron", "Delila", "Samson"],  
"cartography": ["Christopher", "Juan", "Marco"], "philosophy": ["Frederica",  
"Manuel"]}
```

```
person = {"name": "Shuri", "age": 18, "family": ["T'Chaka", "Ramonda"]}
```

Invalid Keys

```
powers = {[1, 2, 4, 8, 16]: 2, [1, 3, 9, 27, 81]: 3}
```

```
TypeError: unhashable type: 'list'
```



Adding / Changing
Values
&
Multiple Values

```
dictionary[key] = value
```

```
menu = {"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry  
muffin": 2}
```

```
menu["cheesecake"] = 8
```

```
{"oatmeal": 3, "avocado toast": 6, "carrot juice": 5, "blueberry muffin": 2,  
"cheesecake": 8}
```

```
sensors = {"living room": 21, "kitchen": 23, "bedroom": 20}

sensors.update({"pantry": 22, "guest room": 25, "patio": 34})

{"living room": 21, "kitchen": 23, "bedroom": 20, "pantry": 22, "guest
room": 25, "patio": 34}
```

Dict Comprehensions

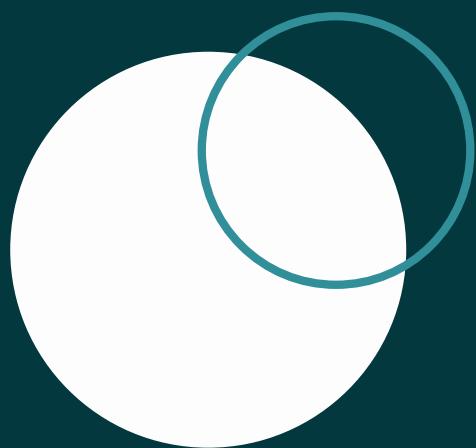
```
names = ['Jenny', 'Alexus', 'Sam', 'Grace']
heights = [61, 70, 67, 64]
```

```
students = {key:value for key, value in zip(names, heights)}
#students is now {'Jenny': 61, 'Alexus': 70, 'Sam': 67, 'Grace': 64}
```

CLASS

```
class CoolClass:  
    pass
```

Una clase es una plantilla para un tipo de datos. Describe los tipos de información que contendrá la clase y cómo un programador interactuará con esos datos. Defina una clase usando la `class` palabra clave.

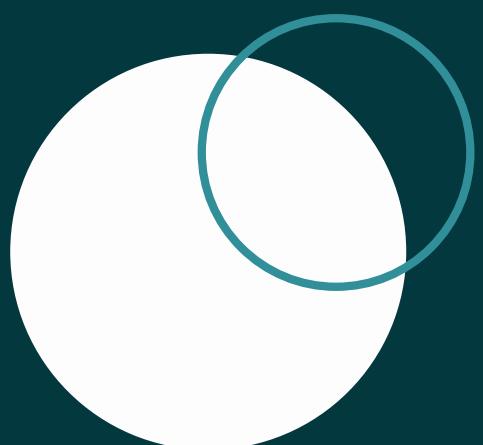


La Guía de estilo de PEP 8 para código Python recomienda poner en mayúscula los nombres de las clases para que sean más fáciles de identificar.

OOP

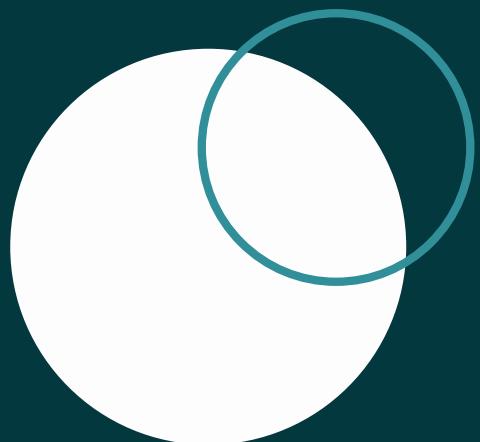
```
print(type(cool_instance))  
# prints "<class '__main__.CoolClass'>"
```

La creación de instancias toma una clase y la convierte en un objeto, la type() función hace lo contrario. Cuando se llama con un objeto, devuelve la clase de la que el objeto es una instancia.



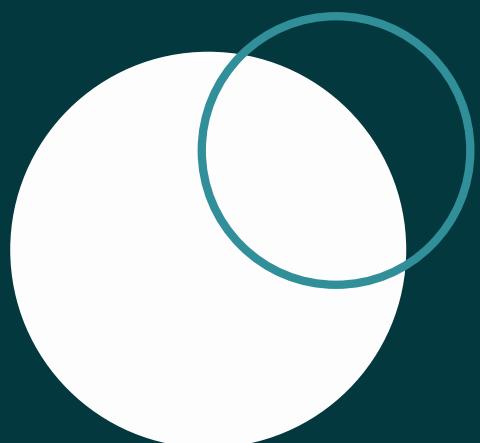
CONSTRUCTORS

La palabra "constructor" se usa para describir características similares en otros lenguajes de programación orientados a objetos, pero los programadores que se refieren a un constructor en Python generalmente se refieren al `__init__()` método.



`__init__(self)`

El primer método de dunder que vamos a utilizar es el `__init__()`método (tenga en cuenta los dos guiones bajos antes y después de la palabra "init"). Este método se utiliza para inicializar un objeto recién creado. Se llama cada vez que se instancia la clase.



```
class Shouter:  
    def __init__(self):  
        print("HELLO?!")
```

```
shout1 = Shouter()  
# prints "HELLO?!"
```

```
shout2 = Shouter()  
# prints "HELLO?!"
```

```
class Shouter:  
    def __init__(self, phrase):  
        # make sure phrase is a string  
        if type(phrase) == str:  
            # then shout it out  
            print(phrase.upper())
```

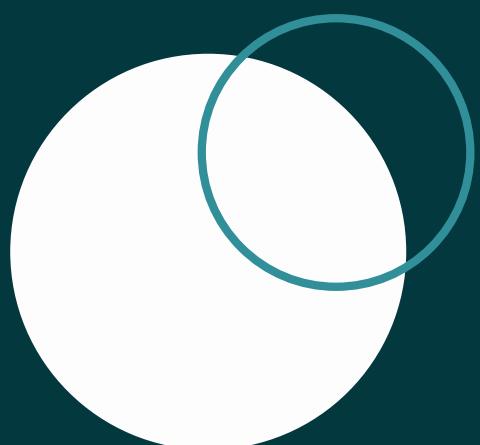
```
shout1 = Shouter("shout")  
# prints "SHOUT"
```

```
shout2 = Shouter("shout")  
# prints "SHOUT"
```

```
shout3 = Shouter("let it all out")  
# prints "LET IT ALL OUT"
```

self

Dado que ya podemos usar diccionarios para almacenar pares clave-valor, usar objetos para ese propósito no es realmente útil. Las variables de instancia son más poderosas cuando puede garantizar la rigidez de los datos que contiene el objeto.



self

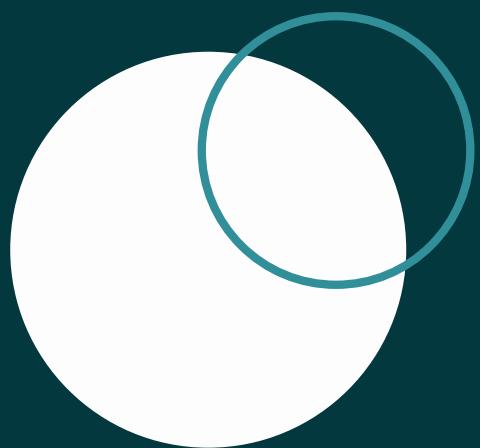
Esta conveniencia es más evidente cuando el constructor crea las variables de instancia, utilizando los argumentos que se le pasan. Si estuviéramos creando un motor de búsqueda y quisiéramos crear clases para cada entrada separada, podríamos devolver. Lo haríamos así:



```
class SearchEngineEntry:  
    def __init__(self, url):  
        self.url = url  
  
codecademy = SearchEngineEntry("www.codecademy.com")  
wikipedia = SearchEngineEntry("www.wikipedia.org")  
  
print(codecademy.url)  
# prints "www.codecademy.com"  
  
print(wikipedia.url)  
# prints "www.wikipedia.org"
```

self

Dado que la -self- palabra clave se refiere al objeto y no a la clase a la que se llama, podemos definir un secure método en la SearchEngineEntryclase que devuelva el enlace seguro a una entrada.



```
class SearchEngineEntry:  
    secure_prefix = "https://"  
    def __init__(self, url):  
        self.url = url  
  
    def secure(self):  
        return "{prefix}{site}".format(prefix=self.secure_prefix, site=self.url)
```

```
codecademy = SearchEngineEntry("www.codecademy.com")
```

```
wikipedia = SearchEngineEntry("www.wikipedia.org")
```

```
print(codecademy.secure())
```

```
# prints "https://www.codecademy.com"
```

```
print(wikipedia.secure())
```

```
# prints "https://www.wikipedia.org"
```

self

Arriba definimos nuestro `secure()` método para tomar solo el único argumento requerido, `self`.

Accedemos tanto a la variable de clase como a `self.secure_prefix` la variable de instancia `self.url` para devolver una URL segura.

