TABLA DE CONTENIDO

EL ZEN DE PYTHON

Hermoso es mejor que feo.
Explícito es mejor que implícito.
Simple es mejor que complejo.
Complejo es mejor que complicado.
Plano es mejor que anidado.
Escaso es mejor que denso.
La legibilidad cuenta.
Los casos especiales no son lo suficientemente especiales para romper las reglas.
Lo práctico supera a la pureza.
Los errores no deben pasar en silencio.
A menos que sean silenciados.
En cara a la ambigüedad, rechazar la tentación de adivinar.
Debe haber una - y preferiblemente sólo una - manera obvia de hacerlo.
Aunque esa manera puede no ser obvia en un primer momento a menos que seas holandés.
Ahora es mejor que nunca.
Aunque "nunca" es a menudo mejor que "ahora mismo".
Si la aplicación es difícil de explicar, es una mala idea.
Si la aplicación es fácil de explicar, puede ser una buena idea.
Los espacios de nombres son una gran idea ¡hay que hacer más de eso!

CONCEPTOS BÁSICOS DEL LENGUAJE

1. Guía de instalación

Python es un lenguaje de programación creado por Guido Van Rossum, con una sintaxis muy limpia, ideado para enseñar a la gente a programar bien. Se trata de un lenguaje interpretado o de script.

1.1. Ventajas de python

- Legible: sintaxis intuitiva y estricta.
- Productivo: ahorra mucho código.
- Portable: para todo sistema operativo.
- Recargado: viene con muchas librerías por defecto.
- Editor recomendado: Atom o Sublime Text.

1.2. Instalación de python

Existen dos versiones de Python que tienen gran uso actualmente, Python 2.x y Python 3.x, para este curso necesitas usar una versión 3.x. Para instalar Python solo debes seguir los pasos dependiendo del sistema operativo que tengas instalado.

1.2.1. Windows

Para instalar Python en Windows ve al sitio https://www.python.org/downloads/ y presiona sobre el botón Download Python 3.7.3. Se descargará un archivo de instalación con el nombre python-3.7.3.exe , ejecútalo. Y sigue los pasos de instalación. Al finalizar la instalación haz lo siguiente para corroborar una instalación correcta:

- Presiona las teclas Windows + R para abrir la ventana de Ejecutar.
- Una vez abierta la ventana Ejecutar escribe el comando cmd y presiona ctrl+shift+enter para ejecutar una línea de comandos con permisos de administrador.
- Windows te preguntará si quieres abrir el Procesador de comandos de Windows con permisos de administrador, presiona sí.
- En la línea de comandos escribe:

```
python
```

1.2.2. MacOS

La forma sencilla es tener instalado homebrew y usar el comando: ** Para instalar la Versión 2.7**

```
brew install python
```

Para instalar la Versión 3.x

```
brew install python3
```

1.2.3. Linux

Generalmente Linux ya lo trae instalado, para comprobarlo puedes ejecutar en la terminal el comando Versión 2.7

```
python -v
```

Versión 3.x

```
python3 -v
```

Si el comando arroja un error quiere decir que no lo tienes instalado, en ese caso los pasos para instalarlo cambian un poco de acuerdo con la distribución de linux que estés usando. Generalmente el gestor de paquetes de la distribución de Linux tiene el paquete de Python. Si eres usuario de Ubuntu o Debian por ejemplo puedes usar este comando para instalar la versión 3.1:

```
$ sudo apt-get install python3.1
```

Si eres usuario de Red Hat o Centos por ejemplo puedes usar este comando para instalar python

```
$ sudo yum install python
```

Si usas otra distribución o no has podido instalar Python, o si eres usuario habitual de linux también puedes descargar los archivos para instalarlo manualmente.

2. Conceptos básicos

2.1. Antes de empezar:

Para usar Python debemos tener un editor de texto abierto y una terminal o cmd (línea de comandos en Windows) como administrador. No le tengas miedo a la consola, la consola es tu amiga. Para ejecutar Python abre la terminal y escribe:

```
python
```

Te abrirá una consola de Python, lo notarás porque el prompt cambia y ahora te muestra tres símbolos de mayor que " >>> " y el puntero adelante indicando que puedes empezar a ingresar comandos de Python:

```
>>>
```

En este modo puedes usar todos los comandos de Python o escribir código directamente. Para salir de Python y regresar a la terminal debes usar el comando:

```
>>> exit()
```

Si deseas ejecutar código de un archivo sólo debes guardarlo con extension.py y luego ejecutar en la terminal:

```
python archivo.py
```

- Uniwebsidad. Python para principiantes
- Ver ejemplo de imprimir en pantalla

Ten en cuenta que para ejecutar el archivo con extensión ".py" debes estar ubicado en el directorio donde tienes guardado el archivo.

Cuando usamos Python debemos atender ciertas reglas de la comunidad para definir su estructura. Las encuentras en el libro PEP8 (Ver enlace).

2.2. Variables

Las variables, a diferencia de los demás lenguajes de programación, no debes definirlas, ni tampoco su tipo de dato, ya que al momento de iterarlas se identificará su tipo. Recuerda que en Python todo es un objeto.

```
A = 3
B = A
```

- Ver ejemplo de variables

2.3. Tipos de datos

2.3.1. Cadenas (str)

Unión de caracteres, palabras o frases. Sintáxis: “Hola”, “¿Cómo estás?”. Ver ejemplo de cadenas

2.3.2. Enteros (int)

En este grupo están todos los números, enteros y long. Sintaxis: 1, 2.3, 2121, 2192, -123. Ver ejemplo de enteros.

- Operadores de asignación

2.3.3. Booleanos (bool)

Son los valores falso o verdadero, compatibles con todas las operaciones lógicas booleanas (and, not, or). Sintaxis: True, False

- Ver ejemplo de tipos de datos

2.4. Condicional if

Ten en cuenta que lo que contiene los paréntesis es la comparación que debe cumplir para que los elementos se cumplan. Los condicionales tienen la siguiente estructura.

```
if ( a > b ):
    elementos
elif ( a == b ):
    elementos
```

```
else:
    elementos
```

2.5. Bucle while

Son un grupo o array de datos, puede contener cualquiera de los datos anteriores. Sintaxis: [1,2,3, ”hola” , [1,2,3]], [1,“Hola”,True]

2.6. Bucle for

Son un grupo o array de datos, puede contener cualquiera de los datos anteriores. Sintaxis: [1,2,3, ”hola” , [1,2,3]], [1,“Hola”,True]

2.7. Estructuras de datos

2.7.1. Listas (list)

Son un grupo o array de datos, puede contener cualquiera de los datos anteriores. Sintaxis: [1,2,3, ”hola” , [1,2,3]], [1,“Hola”,True]

2.7.2. Tuplas (tuple)

También son un grupo de datos igual que una lista con la diferencia que una tupla después de creada no se puede modificar. Sintaxis: (1,2,3, ”hola” , (1,2,3)), (1,“Hola”,True). Sin embargo, jamás podremos cambiar los elementos dentro de esa Tupla.

2.7.3. Diccionarios (dict)

Son un grupo de datos que se acceden a partir de una clave. En los diccionarios tienes un grupo de datos con un formato: la primera cadena o número será la clave para acceder al segundo dato, el segundo dato será el dato al cual accederás con la llave. Recuerda que los diccionarios son listas de llave:valor. Sintaxis: {“clave”:”valor”}, {“nombre”:”Fernando”}

- Ver ejemplo de estructuras de datos

2.8. Funciones

Las funciones las defines con “def” junto a un nombre y unos paréntesis que reciben los parámetros a usar y terminas con dos puntos (:). Sintaxis: def nombre_de_la_función(parametros):

- Ver ejemplo de funciones
- Alcance de variables: Global y Local

2.9. Conversiones

2.9.1. De flotante a entero:

```
>>> int(4.3)
```

4

2.9.2. De entero a flotante:

```
>>> float(4)
```

4.0

2.9.3. De entero a string:

```
>>> str(4.3)
```

"4.3"

2.9.4. De tupla a lista:

```
>>> list((4, 5, 2))
```

[4, 5, 2]

2.10. Operadores Comunes

2.10.1. Longitud de una cadena, lista, tupla, etc.:

```
>>> len("key")
```

3

2.10.2. Tipo de dato:

```
>>> type(4)
```

< type int >

2.10.3. Aplicar una conversión a un conjunto como una lista:

```
>>> map(str, [1, 2, 3, 4])
```

['1', '2', '3', '4']

2.10.4. Redondear un flotante con x número de decimales:

```
>>> round(6.3243, 1)
6.3
```

2.10.5. Generar un rango en una lista (esto es mágico):

```
>>> range(5)
[0, 1, 2, 3, 4]
```

2.10.6. Sumar un conjunto:

```
>>> sum([1, 2, 4])
7
```

2.10.7. Organizar un conjunto:

```
>>> sorted([5, 2, 1])
[1, 2, 5]
```

2.10.8. Conocer los comandos que le puedes aplicar a x tipo de datos:

```
>>>Li = [5, 2, 1]
>>>dir(Li)
>>>['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort' son posibles comandos que puedes aplicar a una lista.

2.10.9. Información sobre una función o librería:

```
>>> help(sorted)
```

(Aparecerá la documentación de la función sorted)

2.11. Clases

Clases es uno de los conceptos con más definiciones en la programación, pero en resumen sólo son la representación de un objeto. Para definir la clase usas_ class_ y el nombre. En caso de tener parámetros los pones entre paréntesis.

Para crear un constructor haces una función dentro de la clase con el nombre init y de parámetros self (significa su clase misma), nombre_r y edad_r:

```
>>>  class Estudiante(object):
...      def __init__(self,nombre_r,edad_r):
...          self.nombre = nombre_r
...          self.edad = edad_r
...
...      def hola(self):
...          return "Mi nombre es %s y tengo %i" % (self.nombre, self.edad)
...
>>> e = Estudiante(“Arturo”, 21)
>>> print (e.hola())
```

Mi nombre es Arturo y tengo 21

Lo que hicimos en las dos últimas líneas fue:

- En la variable e llamamos la clase Estudiante y le pasamos la cadena “Arturo” y el entero 21.
- Imprimimos la función hola() dentro de la variable e (a la que anteriormente habíamos pasado la clase).

Y por eso se imprime la cadena “Mi nombre es Arturo y tengo 21”

2.12. Métodos especiales

- cmp(self,otro)
  Método llamado cuando utilizas los operadores de comparación para comprobar si tu objeto es menor, mayor o igual al objeto pasado como parámetro.

- len(self)
  Método llamado para comprobar la longitud del objeto. Lo usas, por ejemplo, cuando llamas la función len(obj) sobre nuestro código. Como es de suponer el método te debe devolver la longitud del objeto.

- init(self,otro)
  Es un constructor de nuestra clase, es decir, es un “método especial” que se llama automáticamente cuando creas un objeto.

2.13. Condicionales IF

2.14. Bucle FOR

El bucle de for lo puedes usar de la siguiente forma: recorres una cadena o lista a la cual va a tomar el elemento en cuestión con la siguiente estructura:

```
for i in ____:
    elementos
```

Ejemplo:

```
for i in range(10):
    print (i)
```

En este caso recorrerá una lista de diez elementos, es decir el _print i _de ejecutar diez veces. Ahora i va a tomar cada valor de la lista, entonces este for imprimirá los números del 0 al 9 (recordar que en un range vas hasta el número puesto -1).

2.15. Bucle WHILE

En este caso while tiene una condición que determina hasta cuándo se ejecutará. O sea que dejará de ejecutarse en el momento en que la condición deje de ser cierta. La estructura de un while es la siguiente:

```
while (condición):
    elementos
```

Ejemplo:

```
>>> x = 0
>>> while x < 10:
... print x
... x += 1
```

En este ejemplo preguntará si es menor que diez. Dado que es menor imprimirá x y luego sumará una unidad a x. Luego x es 1 y como sigue siendo menor a diez se seguirá ejecutando, y así sucesivamente hasta que x llegue a ser mayor o igual a 10.

3. Archivos y slides del curso práctico de Python

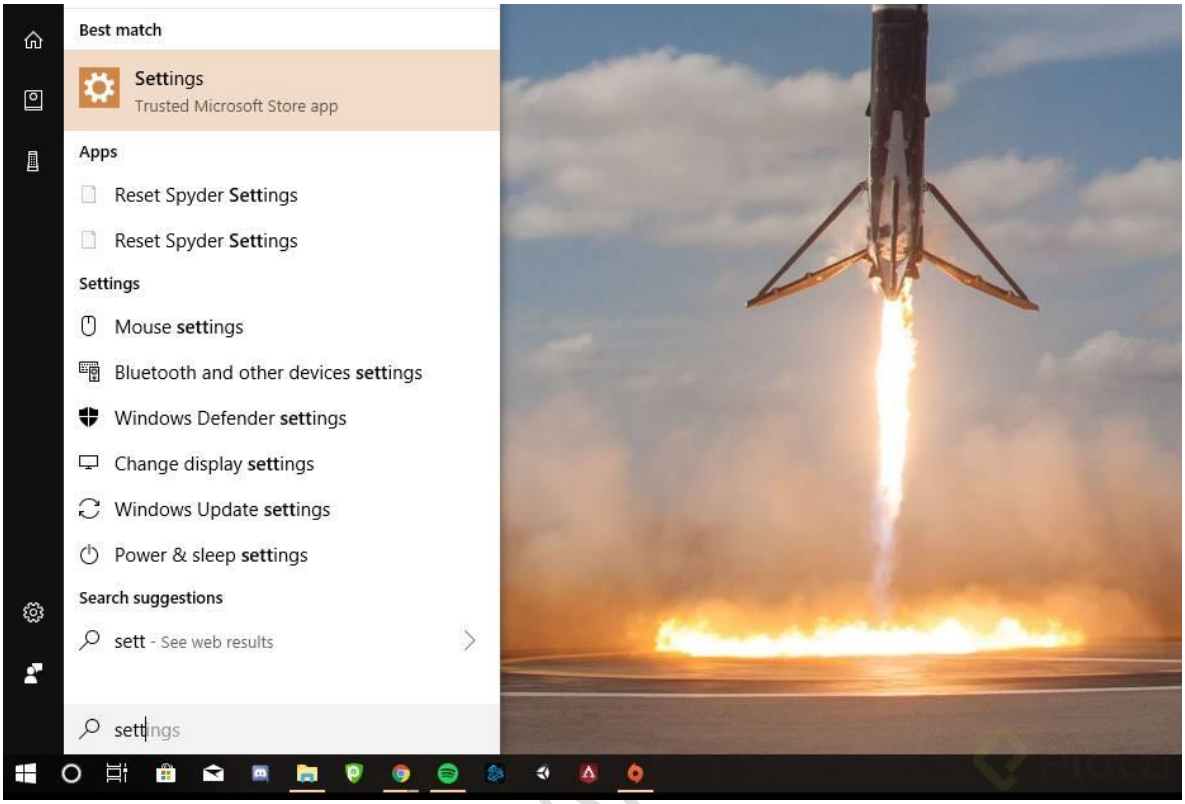
Bienvenida o bienvenido a este nuevo curso de Python 3, en este curso aprenderás los conceptos más importantes del lenguaje a través del desarrollo de un proyecto que funciona como un CRUD utilizando Python 3 puro.

A continuación encontrarás los slides en formato pdf:
https://drive.google.com/file/d/1uAC0egE_U6571mV8gHtHq5ahIbo9vd1e/view?usp=sharing

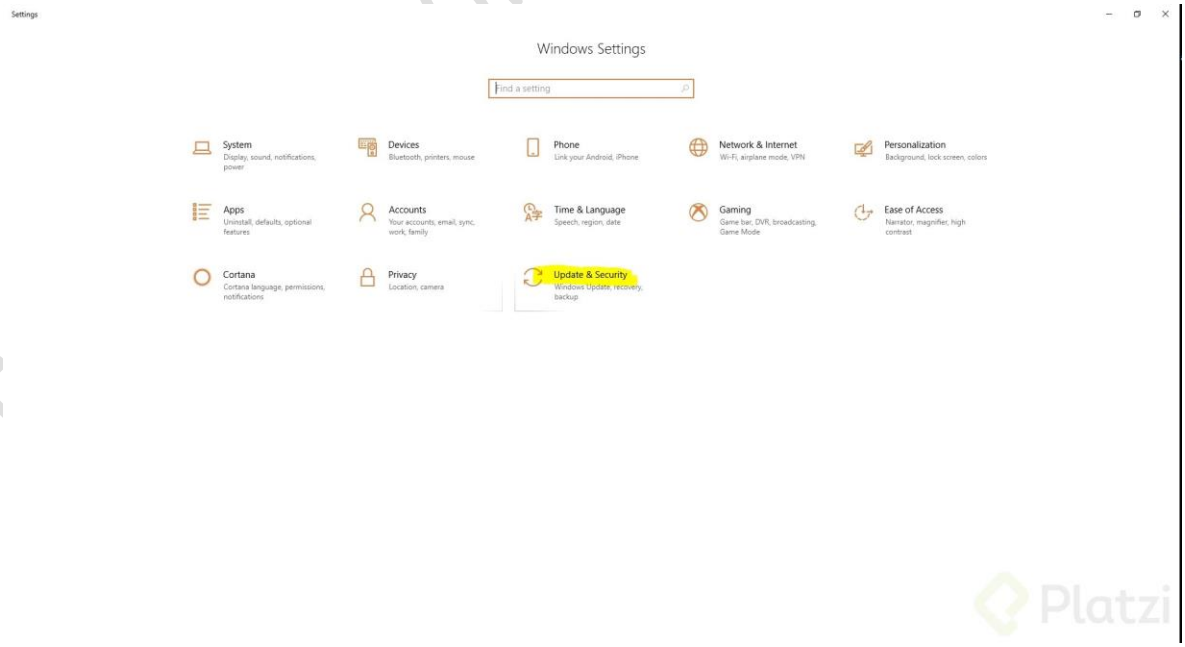
Y también el repositorio completo del curso en el cual encontrarás todo el proyecto dividido en secciones tal como se fue desarrollando:
https://github.com/platzi/curso_Python3/branches

4. Instalando Ubuntu Bash en Windows

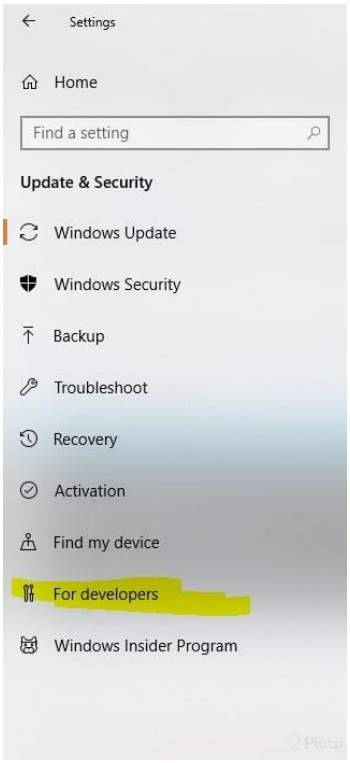
En este tutorial te enseñaré a configurar el Ubuntu dentro de tú Windows 10 para que puedas ejecutar los comandos tal como los ejecuta el profesor en el curso. Lo primero que necesitas es que tu computadora tenga instalado Windows 10 de 64 bits y tengas tu sistema operativo actualizado (con el "Windows 10 Anniversary Update"). Una vez hayas verificado que tu computadora cumple con los requisitos entra a los settings del sistema (Ajustes).

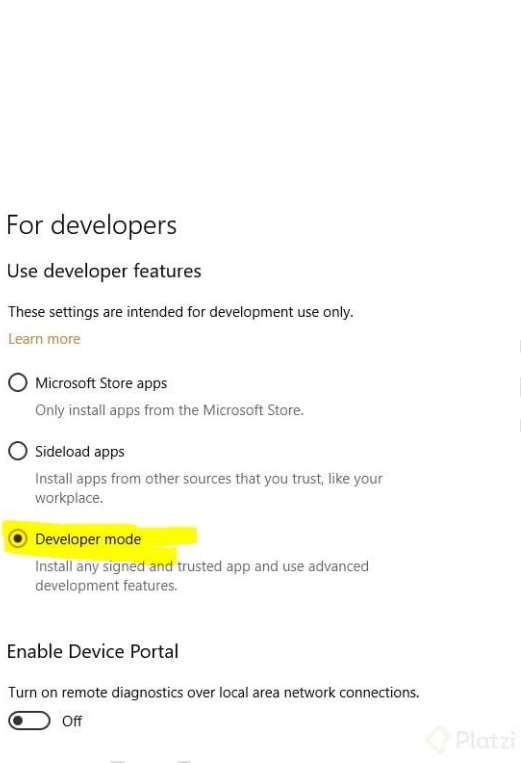


Luego entra a la opción de Actualizaciones y Seguridad

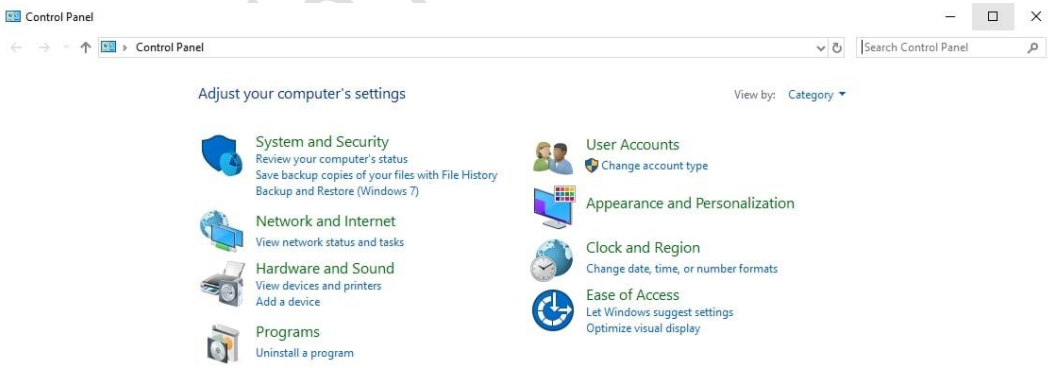


En el menú de la izquierda has click en opciones para desarrolladores y habilita el “Modo Desarrollador”

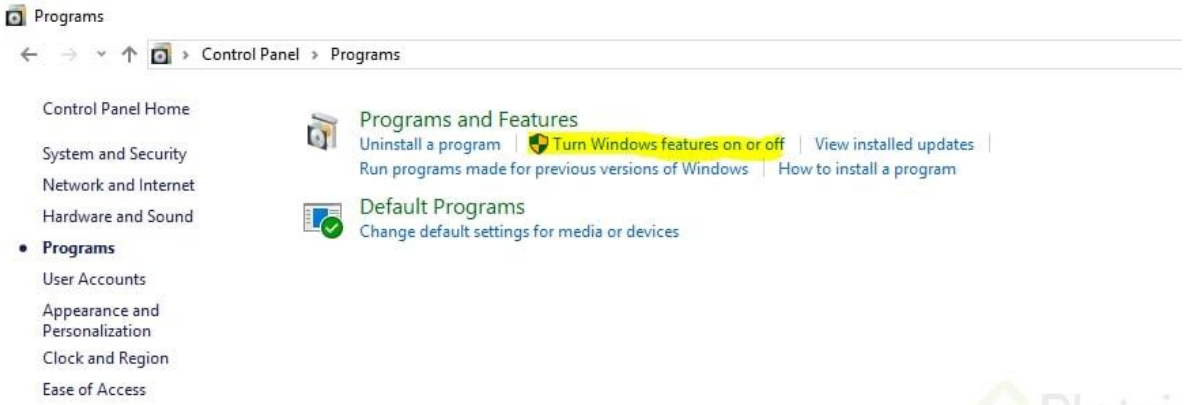




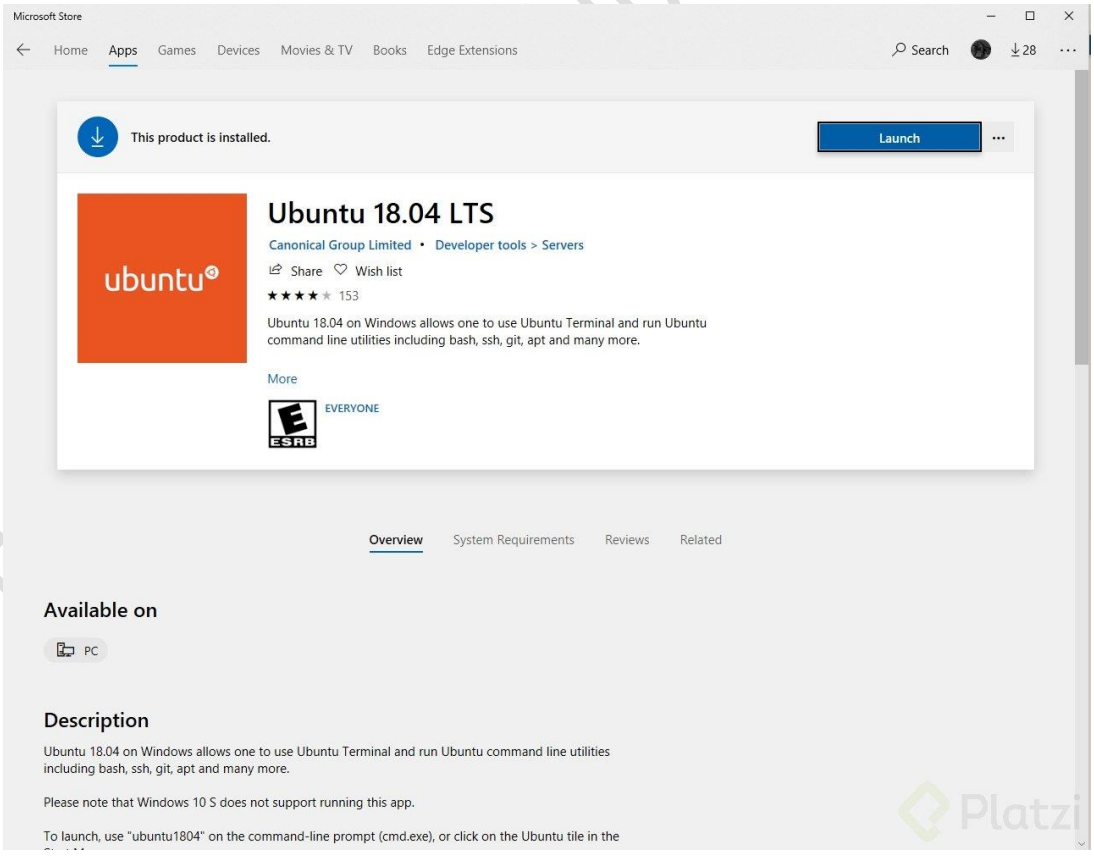
Después, accede al panel de control y haz click en “Programas”



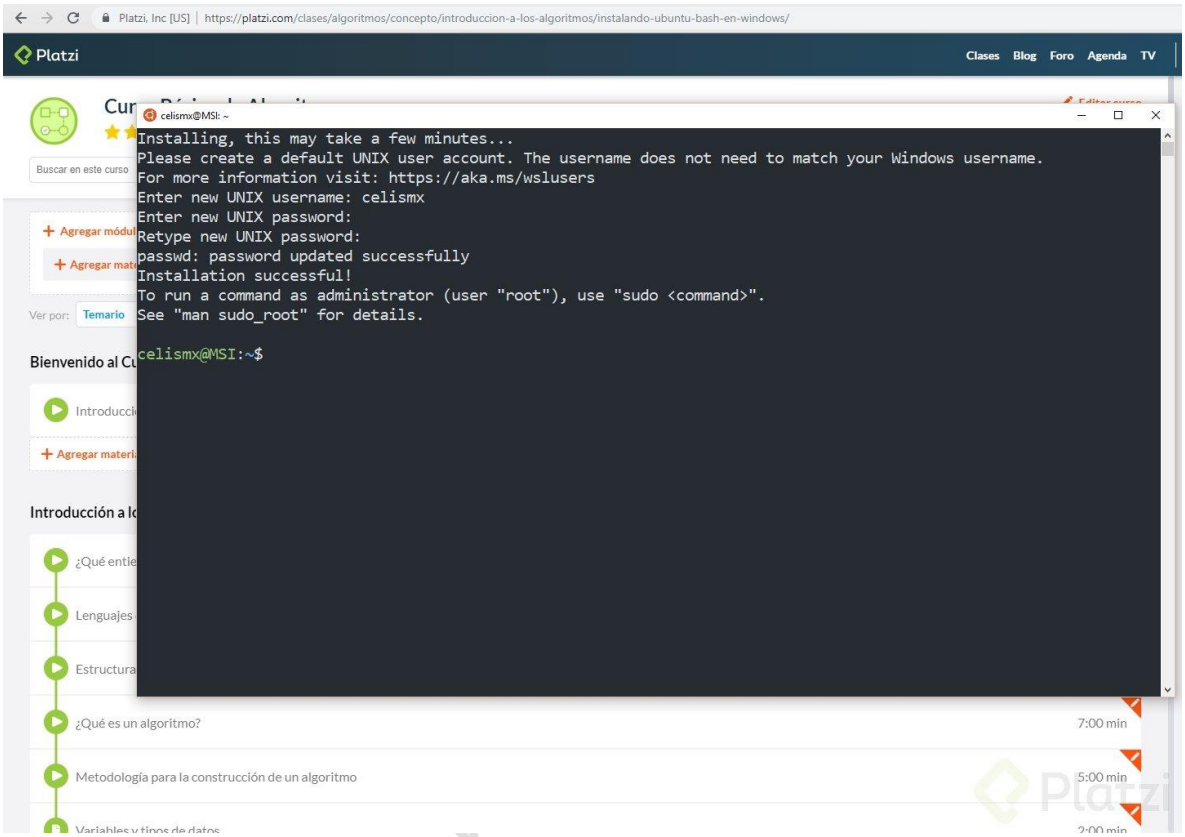
Una vez ahí, haz click en activar o desactivar características de Windows



Aquí, busca la opción de “Windows Subsystem for Linux” y actívala, instala eso y permite que tu computadora se reinicie. Luego, entra al menú inicio, escribe bash y sigue los pasos que te indique, en caso de que te diga que no tienes ninguna distribución sólo ve a la tienda de aplicaciones y descargaba Ubuntu para Windows.



Luego, ejecuta Ubuntu, crea tu usuario y contraseña y estás lista o listo para continuar.



por último, instala Python usando

```
sudo apt-get update
```

y luego ejecuta

```
sudo apt-get install python3
```

una vez termine la instalación, prueba ejecutando “python3”.

Bonus: para moverte a tus carpetas en tu disco duro usa el siguiente comando:

```
cd ../../mnt/c/Users/NOMBREDEUSUARIO/
```

5. ¿Qué es la programación?

Python es uno de los lenguajes más emocionantes de la actualidad y puedes lograr muchas cosas con él. Este curso te va a servir como una introducción al lenguaje. ¿Qué es la programación?

Es una disciplina que combina parte de otras disciplinas como las Matemáticas, Ingeniería y la Ciencia. Sin embargo, la habilidad más importante es resolver problemas. Es lo que harás todos los días como programador o programadora. La programación es una secuencia de instrucciones que le damos a la computadora para que haga lo que nosotros deseamos. Podemos construir una aplicación web, móvil, un programa que lleve cohetes a la luna o marte, resolver problemas de finanzas. La estructura de un programa. Casi todos los programas tienen un input, output, operaciones matemáticas, ejecución condicional y repeticiones.

Objetivos del curso:

- Aprender a pensar como un Científico de la Computación
- Aprender a utilizar Python
- Entender las ventajas y desventajas de Python
- Aprender a construir una aplicación de línea de comandos.

Archivos de la clase: https://github.com/platzi/curso_Python3/tree/1-what-is-programming

6. ¿Por qué programar con Python?

Python es uno de los mejores lenguajes para principiantes porque tiene una sintaxis clara, una gran comunidad y esto hace que el lenguaje sea muy amigable para los que están iniciando. Python está diseñado para ser fácil de usar, a diferencia de otros lenguajes donde la prioridad es ser rápido y eficiente. Python no es de los lenguajes más rápidos, pero casi nunca importa. Es el tercer lenguaje, según Github, entre los más populares. En StackOverflow se comenta que es uno de los lenguajes que mayor popularidad está obteniendo.

““Python cuando podamos, C++ cuando necesitemos””
python --version para conocer la versión que tenemos instalada
python [nombre del archivo] para ejecutar nuestro programa

Archivos de la clase: https://github.com/platzi/curso_Python3/tree/2-why-program-with-python
Lecturas recomendadas:
Welcome to Python.org: https://www.python.org/
Se recomienda trabajar con la librería turtle, para trabajar la lógica: import turtle

7. Operadores matemáticos

En programación estos operadores son muy similares a nuestras clases básicas de matemáticas.

- //: Es división de entero, básicamente tiramos la parte decimal
- %: Es el residuo de la división, lo que te sobra.
- **: Exponente

Los operadores son contextuales, dependen del tipo de valor. Un valor es la representación de una entidad que puede ser manipulada por un programa. Podemos conocer el tipo del valor con type() y nos devolverá algo similar a <class 'init'>, <class 'float'>, <class 'str'>. Dependiendo del tipo los operadores van a funcionar de manera diferente.

8. Variables y expresiones

Una variable es simplemente el contenedor de un valor. Es una forma de decirle a la computadora de que nos guarde un valor para luego usarlo. Python es un lenguaje dinámico, este concepto de privado y público se genera por convenciones del lenguaje. En programación el signo = significa asignación. Si una variable está en mayúscula, usualmente se refiere a una constante, no debería reasignarse. Es una convención.

Reglas de Variables:

- Pueden contener números y letras
- No deben comenzar con número
- Múltiples palabras se unen con _
- No se pueden utilizar palabras reservadas
- El guión bajo en una variable indica que es privada: Ejemplo _age.
- Doble guión bajo, es una variable super privada: Ejemplo __do_not_touch “Si se modifica puede dañar todo”.

Expresiones son instrucciones para que el intérprete evalúe una expresión. Los enunciados tienen efectos dentro del programa, como print que genera un output.

PEMDAS = Paréntesis, Exponente, Multiplicación-División, Adición-Sustracción

Archivos de la clase: https://github.com/platzi/curso_Python3/tree/3-variables-and-expressions

9. Presentación del proyecto

Un vistazo al proyecto de línea de comandos llamado Platzi Ventas la cual nos va a servir para manejar clientes, ventas, inventarios y generar algunos reportes.

- Limpiar terminal

clear

- Desde la línea de comandos podemos crear un archivo con:

touch [archive]
touch main.py

- Escribiremos el primer archivo main.py:

```
1  clients = 'pablo,ricardo,'
2
3  if __name__ == '__main__':
4      clients += 'david'
5      print(clients)
```

- Para saber en qué directorio estamos:

pwb

- Mover un archivo:

mv [archivo] [../carpeta]

- Saber nombre de archivos y carpetas:

ls

10. Funciones

En el contexto de la programación las funciones son simplemente una agrupación de enunciados(statments) que tienen un nombre. Una función tiene un nombre, debe ser descriptivo, puede tener parámetros y puede regresar un valor después que se generó el cómputo.

Python es un lenguaje que se conoce como batteries include(baterías incluidas) esto significa que tiene una librería estándar con muchas funciones y librerías. Para declarar funciones que no son las globales, las built-in functions, necesitamos importar un módulo.

Con el keyword def declaramos una función.

11. Usando Funciones en nuestro proyecto

- Modificamos el archivo main.py:

```
1  clients = 'pablo,ricardo,'
2
3  def create_client(client_name):
4      global clients
5
6      clients += client_name
7      _add_comma()
8
9
10 def list_clients():
11     global clients
12
13     print(clients)
14
15
16 def _add_comma():
17     global clients
18
19     clients += ','
20
21 if __name__ == '__main__':
22     list_clients()
23     create_client('david')
24     list_clients()
```

Lectura recomendada: https://static.platzi.com/media/public/uploads/lambdas_09e88ca0-df9a-4098-b475-9b9b6d0f4d7a.pdf

12. Operadores lógicos

Para comprender el flujo de nuestro programa debemos entender un poco sobre estructuras y expresiones booleanas

- == se refiere a igualdad
- != no hay igualdad.
- > mayor que
- < menor que

- >= mayor o igual
- <= menor o igual

- and unicamente es verdadero cuando ambos valores son verdaderos
- or es verdadero cuando uno de los dos valores es verdadero.
- not es lo contrario al valor. Falso es Verdadero. Verdadero es Falso.

Curso recomendado: Curso de matemáticas discretas (Materiales)
https://platzi.com/cursos/discretas/

13. Estructuras condicionales

En esta clase seguiremos construyendo nuestro proyecto PlatziVentas haciéndolo un poco más interesante y conoceremos un poco sobre las Estructuras condicionales. En Python es importante importante la indentación, de esa manera identifica donde empieza y termina un bloque de código sin necesidad de llaves {} como en otros lenguajes.

- Modificamos el archivo main.py:

```
1  clients = 'pablo,ricardo,'
2
3  def create_client(client_name):
4      global clients
5
6      if client_name not in clients:
7          clients += client_name
8          _add_comma()
9      else:
10         print('Client already is in the client\'s list')
11
12
13 def list_clients():
14     global clients
15
16     print(clients)
17
18
19 def _add_comma():
```

```
20      global clients
21
22      clients += ','
23
24
25  def _print_welcome():
26      print('WELCOME TO PLATZI VENTAS')
27      print('*' * 50)
28      print('What would you like to do today?')
29      print('[C]reate client')
30      print('[D]elete client')
31      print()
32
33
34  if __name__ == '__main__':
35      _print_welcome()
36
37      command = input()
38
39      if command == 'C':
40          client_name = input('What is the client name? ')
41          create_client(client_name)
42          list_clients()
43      elif command == 'D':
44          pass
45      else:
46          print('Invalid command')
47
48  print()
```

USO DE STRINGS Y CICLOS

14. Strings en Python

Los strings o cadenas de textos tienen un comportamiento distinto a otros tipos como los booleanos, enteros, floats. Las cadenas son secuencias de caracteres, todas se pueden acceder a través de un índice. Podemos saber la longitud de un string, cuántos caracteres se encuentran en esa secuencia. Lo podemos saber con la built-in function global llamada len.

Algo importante a tener en cuenta cuando hablamos de strings es que estos son inmutables, esto significa que cada vez que modificamos uno estamos generando un nuevo objeto en memoria. El índice de la primera letra es 0, en la programación se empieza a contar desde 0

15. Operaciones con Strings en Python

Los strings tienen varios métodos que nosotros podemos utilizar.

- upper: convierte todo el string a mayúsculas
- lower: convierte todo el string a minúsculas
- find: encuentra el indice en donde existe un patrón que nosotros definimos
- startswith: significa que empieza con algún patrón.
- endswith: significa que termina con algún patrón
- capitalize: coloca la primera letra en mayúscula y el resto en minúscula
- in y not in nos permite saber con cualquier secuencia sin una subsecuencia o substrings se encuentra adentro de la secuencia mayor.

- dir: Nos dice todos los métodos que podemos utilizar dentro de un objeto.
- help: nos imprime en pantalla el docstrings o comentario de ayuda o instrucciones que posee la función. Casi todas las funciones en Python las tienen.

16. Operaciones con Strings y el comando Update

En esta clase seguiremos construyendo nuestro proyecto PlatziVentas, agregaremos el comando update para poder actualizar nuestros clientes y pondremos en práctica lo aprendido en clases anteriores sobre Strings.

```
clients = 'pablo,ricardo,'

def create_client(client_name):
    global clients

    if client_name not in clients:
        clients += client_name
        _add_comma()
    else:
        print('Client already is in the client\'s list')


def list_clients():
    global clients

    print(clients)


def update_client(client_name, update_client_name):
    global clients

    if client_name in clients:
        clients = clients.replace(client_name + ',', update_client_name + ',')
    else:
        print()
        print('Client is not in client\'s list')


def _add_comma():
    global clients

    clients += ','


def _get_client_name():
    return input('What is the client name?: ')


def _print_welcome():
```

```
40      print('WELCOME TO PLATZI VENTAS')
41      print('*' * 50)
42      print('What would you like to do today?')
43      print('[C]reate client')
44      print('[U]pdate client')
45      print('[D]elete client')
46      print()
47
48
49  if __name__ == '__main__':
50      _print_welcome()
51
52      command = input()
53      command = command.upper()
54      print()
55
56      if command == 'C':
57          client_name = _get_client_name()
58          create_client(client_name)
59          print()
60          list_clients()
61      elif command == 'D':
62          pass
63      elif command == 'U':
64          client_name = _get_client_name()
65          update_client_name = input('What is the update client name?: ')
66          update_client(client_name, update_client_name)
67          print()
68          list_clients()
69      else:
70          print('Invalid command')
71
72  print()
```


17. Operaciones con Strings y el comando Delete

En esta clase seguiremos construyendo nuestro proyecto PlatziVentas, agregaremos el comando delete para poder borrar nuestros clientes y pondremos en práctica lo aprendido en clases anteriores sobre Strings.

```
1  clients = 'pablo,ricardo,'
2
3  def create_client(client_name):
4      global clients
5
6      if client_name not in clients:
7          clients += client_name
8          _add_comma()
9      else:
10         print('Client already is in the client\'s list')
11
12
13 def list_clients():
14     global clients
15
16     print(clients)
17
18
19 def update_client(client_name, update_client_name):
20     global clients
21
22     if client_name in clients:
23         clients = clients.replace(client_name + ',', update_client_name + ',')
24     else:
25         print()
26         print('Client is not in client\'s list')
27
28
29 def delete_client(client_name):
30     global clients
31
32     if client_name in clients:
33         clients = clients.replace(client_name + ',', '')
34     else:
35         print()
36         print('Client is not in client\'s list')
37
38
39 def _add_comma():
```

```
40      global clients
41
42      clients += ','
43
44
45  def _get_client_name():
46      return input('What is the client name?: ')
47
48
49  def _print_welcome():
50      print('WELCOME TO PLATZI VENTAS')
51      print('*' * 50)
52      print('What would you like to do today?')
53      print('[C]reate client')
54      print('[L]ist clients')
55      print('[U]pdate client')
56      print('[D]elete client')
57      print()
58
59
60  if __name__ == '__main__':
61      _print_welcome()
62
63      command = input()
64      command = command.upper()
65      print()
66
67      if command == 'C':
68          client_name = _get_client_name()
69          create_client(client_name)
70          print()
71          list_clients()
72      elif command == 'L':
73          list_clients()
74      elif command == 'D':
75          client_name = _get_client_name()
76          delete_client(client_name)
77          print()
78          list_clients()
79      elif command == 'U':
```

```
80          client_name = _get_client_name()
81          update_client_name = input('What is the update client name?: ')
82          update_client(client_name, update_client_name)
83          print()
84          list_clients()
85      else:
86          print('Invalid command')
87
88
89  print()
```

18. Operaciones con Strings: Slices en Python

Los slices en Python nos permiten manejar secuencias de una manera poderosa. Slices en español significa ""rebanada"", si tenemos una secuencia de elementos y queremos una rebanada tenemos una sintaxis para definir qué pedazos queremos de esa secuencia.

secuencia[comienzo:final:pasos]

19. For loops

Las iteraciones es uno de los conceptos más importantes en la programación. En Python existen muchas maneras de iterar pero las dos principales son los for loops y while loops. Los for loops nos permiten iterar a través de una secuencia y los while loops nos permiten iterara hasta cuando una condición se vuelva falsa.

- Tienen dos keywords break y continue que nos permiten salir anticipadamente de la iteración
- Se usan cuando se quiere ejecutar varias veces una o varias instrucciones.
- for [variable] in [secuencia]:

Es una convención usar la letra i como variable en nuestro for, pero podemos colocar la que queramos.

- range: Nos da un objeto rango, es un iterador sobre el cual podemos generar secuencias.

```
1  clients = 'pablo,ricardo,'
2
3  def create_client(client_name):
4      global clients
5
6      if client_name not in clients:
7          clients += client_name
8          _add_comma()
9      else:
10         print('Client already is in the client\'s list')
11
12
13 def list_clients():
14     global clients
15
16     print(clients)
17
18
19 def update_client(client_name, update_client_name):
20     global clients
21
22     if client_name in clients:
23         clients = clients.replace(client_name + ',', update_client_name + ',')
24     else:
25         print()
26         print('Client is not in client\'s list')
27
28
29 def delete_client(client_name):
30     global clients
31
32     if client_name in clients:
33         clients = clients.replace(client_name + ',', '')
34     else:
35         print()
36         print('Client is not in client\'s list')
37
38
39 def search_client(client_name):
```

```
40      global clients
41
42      client_list = clients.split(',')
43
44      for client in client_list:
45          if client != client_name:
46              continue
47          else:
48              return True
49
50
51  def _add_comma():
52      global clients
53
54      clients += ','
55
56
57  def _get_client_name():
58      return input('What is the client name?: ')
59
60
61  def _print_welcome():
62      print('WELCOME TO PLATZI VENTAS')
63      print('*' * 50)
64      print('What would you like to do today?')
65      print('[C]reate client')
66      print('[L]ist clients')
67      print('[U]pdate client')
68      print('[D]elete client')
69      print('[S]earch client')
70      print()
71
72
73  if __name__ == '__main__':
74      _print_welcome()
75
76      command = input()
77      command = command.upper()
78      print()
79
```

```
80      if command == 'C':
81          client_name = _get_client_name()
82          create_client(client_name)
83          print()
84          list_clients()
85      elif command == 'L':
86          list_clients()
87      elif command == 'U':
88          client_name = _get_client_name()
89          update_client_name = input('What is the update client name?: ')
90          update_client(client_name, update_client_name)
91          print()
92          list_clients()
93      elif command == 'D':
94          client_name = _get_client_name()
95          delete_client(client_name)
96          print()
97          list_clients()
98      elif command == 'S':
99          client_name = _get_client_name()
100         found = search_client(client_name)
101         print()
102         if found:
103             print('The client is in our client\'s list')
104         else:
105             print('The client: {} is not in our client\'s list'.format(client_name))
106     else:
107         print('Invalid command')
108
109
110 print()
```


20. While loops

Al igual que las for loops, las while loops nos sirve para iterar, pero las for loops nos sirve para iterar a lo largo de una secuencia mientras que las while loops nos sirve para iterar mientras una condición sea verdadera. Si no tenemos un mecanismo para convertir el mecanismo en falsedad, entonces nuestro while loops se ira al infinito(infinite loop).

```
1  import sys
2
3  clients = 'pablo,ricardo,'
4
5  def create_client(client_name):
6      global clients
7
8      if client_name not in clients:
9          clients += client_name
10         _add_comma()
11     else:
12         print('Client already is in the client\'s list')
13
14
15 def list_clients():
16     global clients
17
18     print(clients)
19
20
21 def update_client(client_name, update_client_name):
22     global clients
23
24     if client_name in clients:
25         clients = clients.replace(client_name + ',', update_client_name + ',')
26     else:
27         print()
28         print('Client is not in client\'s list')
29
30
31 def delete_client(client_name):
32     global clients
33
34     if client_name in clients:
35         clients = clients.replace(client_name + ',', '')
36     else:
37         print()
38         print('Client is not in client\'s list')
39
```

```
40
41  def search_client(client_name):
42      global clients
43
44      client_list = clients.split(',')
45
46      for client in client_list:
47          if client != client_name:
48              continue
49          else:
50              return True
51
52
53  def _add_comma():
54      global clients
55
56      clients += ','
57
58
59  def _get_client_name():
60      client_name = None
61
62      while not client_name:
63          client_name = input('What is the client name?: ')
64
65          if client_name == 'exit':
66              client_name = None
67              break
68
69      if not client_name:
70          print()
71          sys.exit()
72
73      return client_name
74
75
76  def _print_welcome():
77      print('WELCOME TO PLATZI VENTAS')
78      print('*' * 50)
79      print('What would you like to do today?')
```

```
80      print('[C]reate client')
81      print('[L]ist clients')
82      print('[U]pdate client')
83      print('[D]elete client')
84      print('[S]earch client')
85      print()
86
87
88  if __name__ == '__main__':
89      _print_welcome()
90
91      command = input()
92      command = command.upper()
93      print()
94
95      if command == 'C':
96          client_name = _get_client_name()
97          create_client(client_name)
98          print()
99          list_clients()
100     elif command == 'L':
101         list_clients()
102     elif command == 'U':
103         client_name = _get_client_name()
104         update_client_name = input('What is the update client name?: ')
105         update_client(client_name, update_client_name)
106         print()
107         list_clients()
108     elif command == 'D':
109         client_name = _get_client_name()
110         delete_client(client_name)
111         print()
112         list_clients()
113     elif command == 'S':
114         client_name = _get_client_name()
115         found = search_client(client_name)
116         print()
117         if found:
118             print('The client is in our client\'s list')
119         else:
```

```
120             print('The client: {} is not in our client\'s list'.format(client_name))
121     else:
122         print('Invalid command')
123
124
125 print()
```

21. Iterators and generators

Aunque no lo sepas, probablemente ya utilices iterators en tu vida diaria como programador de Python. Un iterator es simplemente un objeto que cumple con los requisitos del Iteration Protocol (protocolo de iteración) y por lo tanto puede ser utilizado en ciclos. Por ejemplo,

```
for i in range(10):
    print(i)
```

En este caso, la función range es un iterable que regresa un nuevo valor en cada ciclo. Para crear un objeto que sea un iterable, y por lo tanto, implemente el protocolo de iteración, debemos hacer tres cosas:

- Crear una clase que implemente los métodos iter y next
- iter debe regresar el objeto sobre el cual se iterará
- next debe regresar el siguiente valor y aventar la excepción StopIteration cuando ya no hayan elementos sobre los cual iterar.

Por su parte, los generators son simplemente una forma rápida de crear iterables sin la necesidad de declarar una clase que implemente el protocolo de iteración. Para crear un generator simplemente declaramos una función y utilizamos el keyword yield en vez de return para regresar el siguiente valor en una iteración. Por ejemplo,

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
```

Es importante recalcar que una vez que se ha agotado un generator ya no podemos utlizarlo y debemos crear una nueva instancia. Por ejemplo,

```
fib1 = fibonacci(20)
fib_nums = [num for num in fib1]
...
double_fib_nums = [num * 2 for num in fib1] # no va a funcionar
double_fib_nums = [num * 2 for num in fibonacci(30)] # sí funciona
```

ESTRUCTURAS DE DATOS

22. Uso de listas

Python y todos los lenguajes nos ofrecen constructos mucho más poderosos, haciendo que el desarrollo de nuestro software sea

- Más sofisticado
- Más legible
- Más fácil de implementar

Estos constructos se llaman Estructuras de Datos que nos permiten agrupar de distintas maneras varios valores y elementos para poderlos manipular con mayor facilidad. Las listas las vas a utilizar durante toda tu carrera dentro de la programación e ingeniería de Software.

Las listas son una secuencia de valores. A diferencia de los strings, las listas pueden tener cualquier tipo de valor. También, a diferencia de los strings, son mutables, podemos agregar y eliminar elementos. En Python, las listas son referenciales. Una lista no guarda en memoria los objetos, sólo guarda la referencia hacia donde viven los objetos en memoria

- Se inician con [] o con la built-in function list.

23. Operaciones con listas

Ahora que ya entiendes cómo funcionan las listas, podemos ver qué tipo de operaciones y métodos podemos utilizar para modificarlas, manipularlas y realizar diferentes tipos de cómputos con esta Estructura de Datos.

- El operador +(suma) concatena dos o más listas.
- El operador *(multiplicación) repite los elementos de la misma lista tantas veces los queramos multiplicar
- Sólo podemos utilizar +(suma) y *(multiplicación).

Las listas tienen varios métodos que podemos utilizar.

- append nos permite añadir elementos a listas. Cambia el tamaño de la lista.
- pop nos permite sacar el último elemento de la lista. También recibe un índice y esto nos permite elegir qué elemento queremos eliminar.

- sort modifica la propia lista y ordenarla de mayor a menor. Existe otro método llamado sorted, que también ordena la lista, pero genera una nueva instancia de la lista
- delnos permite eliminar elementos vía indices, funciona con slices
- remove nos permite es pasarle un valor para que Python compare internamente los valores y determina cuál de ellos hace match o son iguales para eliminarlos.

24. Agregando listas a nuestro proyecto

Agregaremos la reciente Estructura de Datos aprendida a nuestro proyecto de PlatziVentas, en este caso listas.

```
1  import sys
2
3  clients = ['pablo','ricardo']
4
5  def create_client(client_name):
6      global clients
7
8      if client_name not in clients:
9          clients.append(client_name)
10     else:
11         print('Client already is in the client\'s list')
12
13
14 def list_clients():
15     for idx, client in enumerate(clients):
16         print('{}: {}'.format(idx, client))
17
18
19 def update_client(client_name, update_name):
20     global clients
21
22     if client_name in clients:
23         index = clients.index(client_name)
24         clients[index] = update_name
25     else:
26         print()
27         print('Client is not in client\'s list')
28
```

```
29
30  def delete_client(client_name):
31      global clients
32
33      if client_name in clients:
34          clients.remove(client_name)
35      else:
36          print()
37          print('Client is not in client\'s list')
38
39
40  def search_client(client_name):
41      for client in clients:
42          if client != client_name:
43              continue
44          else:
45              return True
46
47
48  def _get_client_name():
49      client_name = None
50
51      while not client_name:
52          client_name = input('What is the client name?: ')
53
54          if client_name == 'exit':
55              client_name = None
56              break
57
58      if not client_name:
59          print()
60          sys.exit()
61
62      return client_name
63
64
65  def _print_welcome():
66      print('WELCOME TO PLATZI VENTAS')
67      print('*' * 50)
68      print('What would you like to do today?')
```

```
69      print('[C]reate client')
70      print('[L]ist clients')
71      print('[U]pdate client')
72      print('[D]elete client')
73      print('[S]earch client')
74      print()
75
76
77  if __name__ == '__main__':
78      _print_welcome()
79
80      command = input()
81      command = command.upper()
82      print()
83
84      if command == 'C':
85          client_name = _get_client_name()
86          create_client(client_name)
87          print()
88          list_clients()
89      elif command == 'L':
90          list_clients()
91      elif command == 'U':
92          client_name = _get_client_name()
93          update_client_name = input('What is the update client name?: ')
94          update_client(client_name, update_client_name)
95          print()
96          list_clients()
97      elif command == 'D':
98          client_name = _get_client_name()
99          delete_client(client_name)
100         print()
101         list_clients()
102     elif command == 'S':
103         client_name = _get_client_name()
104         found = search_client(client_name)
105         print()
106         if found:
107             print('The client is in our client\'s list')
108         else:
```

```
109             print('The client: {} is not in our client\'s list'.format(client_name))
110     else:
111         print('Invalid command')
112
113
114 print()
```

25. Diccionarios

Los diccionarios se conocen con diferentes nombres a lo largo de los lenguajes de programación como HashMaps, Mapas, Objetos, etc. En Python se conocen como Diccionarios. Un diccionario es similar a una lista sabiendo que podemos acceder a través de un indice, pero en el caso de las listas este índice debe ser un número entero. Con los diccionarios puede ser cualquier objeto, normalmente los verán con strings para ser más explicitos, pero funcionan con muchos tipos de llaves.

Un diccionario es una asociación entre llaves(keys) y valores(values) y la referencia en Python es muy precisa. Si abres un diccionario verás muchas palabras y cada palabra tiene su definición.

Para iniciar un diccionario se usa {} o con la función dict

Estos también tienen varios métodos. Siempre puedes usar la función dir para saber todos los métodos que puedes usar con un objeto. Si queremos ciclar a lo largo de un diccionario tenemos las opciones:

- keys: nos imprime una lista de las llaves
- values nos imprime una lista de los valores
- items. nos manda una lista de tuplas de los valores

26. Agregando diccionarios a nuestro proyecto

Agregaremos la reciente Estructura de Datos aprendida a nuestro proyecto de PlatziVentas, en este caso Diccionarios.

```
1  import sys
2
3  clients = [
4      {
```

```
5         'name' : 'pablo',
6         'company' : 'google',
7         'email' : 'pablo@google.com',
8         'position' : 'software enginner',
9     },
10    {
11        'name' : 'ricardo',
12        'company' : 'facebook',
13        'email' : 'ricardo@facebook.com',
14        'position' : 'data science enginner',
15    }
16 ]
17
18 def create_client(client):
19     global clients
20
21     if client not in clients:
22         clients.append(client)
23     else:
24         print('Client already in the client\'s list')
25
26
27 def list_clients():
28     print('-' * 82)
29     print('id | name\t| company\t| email\t\t\t| position\t\t |')
30     print('-' * 82)
31     for idx, client in enumerate(clients):
32         print(' {uid} | {name} \t| {company} \t| {email} \t| {position}\t |'.format(
33             uid = idx,
34             name = client['name'],
35             company = client['company'],
36             email = client['email'],
37             position = client['position']
38         ))
39     print('-' * 82)
40
41
42 def update_client(client_id, update_client):
43     global clients
44
```

```
45      if len(clients) - 1 >= client_id:
46          clients[client_id] = update_client
47      else:
48          print()
49          print('Client not in client\'s list')
50
51
52  def delete_client(client_id):
53      global clients
54
55      for idx, client in enumerate(clients):
56          if idx == client_id:
57              del clients[idx]
58              break
59
60
61  def search_client(client_name):
62      for client in clients:
63          if client['name'] != client_name:
64              continue
65          else:
66              return True
67
68
69  def _get_client_field(field_name, message='What is the client {}?: '):
70      field = None
71
72      while not field:
73          field = input(message.format(field_name))
74
75      return field
76
77
78  def _get_client_from_user():
79      client = {
80          'name': _get_client_field('name'),
81          'company': _get_client_field('company'),
82          'email': _get_client_field('email'),
83          'position': _get_client_field('position'),
84      }
```

```
85
86      return client
87
88
89  def _print_welcome():
90      print('WELCOME TO PLATZI VENTAS')
91      print('*' * 50)
92      print('What would you like to do today?')
93      print('[C]reate client')
94      print('[L]ist clients')
95      print('[U]pdate client')
96      print('[D]elete client')
97      print('[S]earch client')
98      print()
99
100
101 if __name__ == '__main__':
102     _print_welcome()
103
104     command = input()
105     command = command.upper()
106     print()
107
108     if command == 'C':
109         client = {
110             'name': _get_client_field('name'),
111             'company': _get_client_field('company'),
112             'email': _get_client_field('email'),
113             'position': _get_client_field('position'),
114         }
115         create_client(client)
116         print()
117         list_clients()
118     elif command == 'L':
119         list_clients()
120     elif command == 'U':
121         client_id = int(_get_client_field('id'))
122         updated_client = _get_client_from_user()
123
124         update_client(client_id, updated_client)
```

```
125         print()
126         list_clients()
127     elif command == 'D':
128         client_id = int(_get_client_field('id'))
129         delete_client(client_id)
130         print()
131 
132         list_clients()
133     elif command == 'S':
134         client_name = _get_client_field('name')
135         found = search_client(client_name)
136         print()
137         if found:
138             print('The client is in our client\'s list')
139         else:
140             print('The client: {} is not in our client\'s list'.format(client_name))
141     else:
142         print('Invalid command')
143 
144 
145 print()
```

27. Tuplas y conjuntos

Tuplas(tuples) son iguales a las listas, la única diferencia es que son inmutables, la diferencia con los strings es que pueden recibir muchos tipos valores. Son una serie de valores separados por comas, casi siempre se le agregan paréntesis para que sea mucho más legible.

Para poderla inicializar utilizamos la función tuple.

Uno de sus usos muy comunes es cuando queremos regresar más de un valor en nuestra función. Una de las características de las Estructuras de Datos es que cada una de ellas nos sirve para algo especifico. No existe en programación una navaja suiza que nos sirva para todos. Los mejores programas son aquellos que utilizan la herramienta correcta para el trabajo correcto.

Conjutos(sets) nacen de la teoría de conjuntos. Son una de las Estructuras más importantes y se parecen a las listas, podemos añadir varios elementos al conjunto, pero no pueden existir elementos duplicados. A diferencia de los tuples podemos agregar y eliminar, son mutables. Los sets se pueden inicializar con la función set. Una recomendación es inicializarlos con esta función para no causar confusión con los diccionarios.

- add nos sirve añadir elementos.
- remove nos permite eliminar elementos.

28. Tuplas y conjuntos en código

En esta clase practicaremos en código lo aprendido en la clase anterior sobre tuplas(tuples) y conjuntos(sets) para que sea mucho más claro entenderlo.

29. Introducción al módulo collections

El módulo collections nos brinda un conjunto de objetos primitivos que nos permiten extender el comportamiento de las built-in collections que poseé Python y nos otorga estructuras de datos adicionales. Por ejemplo, si queremos extender el comportamiento de un diccionario, podemos extender la clase UserDict; para el caso de una lista, extendemos UserList; y para el caso de strings, utilizamos UserString.

Por ejemplo, si queremos tener el comportamiento de un diccionario podemos escribir el siguiente código:

```
1  class SecretDict(collections.UserDict):
2
3    def _password_is_valid(self, password):
4        …
5
6    def _get_item(self, key):
7        …
8
9    def __getitem__(self, key):
10       password, key = key.split(':')
11
12       if self._password_is_valid(password):
13           return self._get_item(key)
14
15       return None
16
17 my_secret_dict = SecretDict(...)
18 my_secret_dict['some_password:some_key'] # si el password es válido, regresa el valor
```

Otra estructura de datos que vale la pena analizar, es namedtuple. Hasta ahora, has utilizado tuples que permiten acceder a sus valores a través de índices. Sin embargo, en ocasiones es importante poder nombrar elementos (en vez de utilizar posiciones) para acceder a valores y no queremos crear una clase ya que únicamente necesitamos un contenedor de valores y no comportamiento.

```
1  Coffee = collections.NamedTuple('Coffee', ('size', 'bean', 'price'))
2  def get_coffee(coffee_type):
3      If coffee_type == 'houseblend':
4          return Coffee('large', 'premium', 10)
```

El módulo collections también nos ofrece otros primitivos que tienen la labor de facilitarnos la creación y manipulación de colecciones en Python. Por ejemplo, Counter nos permite contar de manera eficiente ocurrencias en cualquier iterable; OrderedDict nos permite crear diccionarios que poseen un orden explícito; deque nos permite crear filas (para pilas podemos utilizar la lista).

En conclusión, el módulo collections es una gran fuente de utilerías que nos permiten escribir código más "pythonico" y más eficiente.

30. Python comprehensions

Las Comprehensions son constructos que nos permiten generar una secuencia a partir de otra secuencia.

Existen tres tipos de comprehensions:

- List comprehensions

  [element for element in element_list if element_meets_condition]

- Dictionary comprehensions

  {key: element for element in element_list if element_meets_condition}

- Sets comprehensions

  {element for element in element_list if elements_meets_condition}

31. Búsquedas binarias

Uno de los conceptos más importantes que debes entender en tu carrera dentro de la programación son los algoritmos. No son más que una secuencia de instrucciones para resolver un problema específico. Búsqueda binaria lo único que hace es tratar de encontrar un resultado en una lista ordenada de tal manera que podamos razonar. Si tenemos un elemento mayor que otro, podemos simplemente usar la mitad de la lista cada vez.

32. Continuando con las búsquedas binarias

```
 1  import random
 2
 3  def binary_search(data, target, low, high) :
 4      if low > high :
 5          return False
 6
 7      mid = (low + high) // 2
 8
 9      if target == data[mid] :
10          return True
11      elif target < data[mid] :
12          return binary_search(data, target, low, mid - 1)
13      else :
14          return binary_search(data, target, mid + 1, high)
15
16
17  if __name__ == '__main__' :
18
19      data = [random.randint(0,100) for i in range(10)]
20      data.sort()
21      print()
22      print(data)
23      print()
24      target = int(input('What number would you like to find?: '))
25      found = binary_search(data, target, 0, len(data) - 1)
26      print()
27      print(found)
28      print()
```

33. Manipulación de archivos en Python 3

```
1  import csv
2  import os
3
4  CLIENT_SCHEMA = ['name', 'company', 'email', 'position']
5  CLIENT_TABLE = '.clients.csv'
6  clients = []
7
8  def create_client(client):
9      global clients
10
11     if client not in clients:
12         clients.append(client)
13     else:
14         print('Client already in the client\'s list')
15
16
17 def list_clients():
18     print('-' * 82)
19     print('id | name\t| company\t| email\t\t\t| position\t\t |')
20     print('-' * 82)
21     for idx, client in enumerate(clients):
22         print(' {uid} | {name} \t| {company} \t| {email} \t| {position}\t |'.format(
23             uid = idx,
24             name = client['name'],
25             company = client['company'],
26             email = client['email'],
27             position = client['position']
28         ))
29     print('-' * 82)
30
31
32 def update_client(client_id, update_client):
33     global clients
34
35     if len(clients) - 1 >= client_id:
36         clients[client_id] = update_client
37     else:
```

```
38          print()
39          print('Client not in client\'s list')
40
41
42  def delete_client(client_id):
43      global clients
44
45      for idx, client in enumerate(clients):
46          if idx == client_id:
47              del clients[idx]
48              break
49
50
51  def search_client(client_name):
52      for client in clients:
53          if client['name'] != client_name:
54              continue
55          else:
56              return True
57
58
59  def _get_client_field(field_name, message='What is the client {}?: '):
60      field = None
61
62      while not field:
63          field = input(message.format(field_name))
64
65      return field
66
67
68  def _get_client_from_user():
69      client = {
70          'name': _get_client_field('name'),
71          'company': _get_client_field('company'),
72          'email': _get_client_field('email'),
73          'position': _get_client_field('position'),
74      }
75
76      return client
77
```

```
78
79  def _initialize_clients_from_storage():
80      with open(CLIENT_TABLE, mode='r') as f:
81          reader = csv.DictReader(f, fieldnames=CLIENT_SCHEMA)
82
83          for row in reader:
84              clients.append(row)
85
86
87  def _save_clients_to_storage():
88      tmp_table_name = '{}.tmp'.format(CLIENT_TABLE)
89      with open(tmp_table_name, mode='w') as f:
90          writer = csv.DictWriter(f, fieldnames=CLIENT_SCHEMA)
91          writer.writerows(clients)
92
93          os.remove(CLIENT_TABLE)
94      os.rename(tmp_table_name, CLIENT_TABLE)
95
96
97  def _print_welcome():
98      print('WELCOME TO PLATZI VENTAS')
99      print('*' * 50)
100     print('What would you like to do today?')
101     print('[C]reate client')
102     print('[L]ist clients')
103     print('[U]pdate client')
104     print('[D]elete client')
105     print('[S]earch client')
106     print()
107
108
109 if __name__ == '__main__':
110     _initialize_clients_from_storage()
111     _print_welcome()
112
113     command = input()
114     command = command.upper()
115     print()
116
117     if command == 'C':
```

```
118         client = {
119             'name': _get_client_field('name'),
120             'company': _get_client_field('company'),
121             'email': _get_client_field('email'),
122             'position': _get_client_field('position'),
123         }
124         create_client(client)
125         print()
126     elif command == 'L':
127         list_clients()
128     elif command == 'U':
129         client_id = int(_get_client_field('id'))
130         updated_client = _get_client_from_user()
131 
132         update_client(client_id, updated_client)
133         print()
134     elif command == 'D':
135         client_id = int(_get_client_field('id'))
136 
137         delete_client(client_id)
138         print()
139     elif command == 'S':
140         client_name = _get_client_field('name')
141         found = search_client(client_name)
142         print()
143         if found:
144             print('The client is in our client\'s list')
145         else:
146             print('The client: {} is not in our client\'s list'.format(client_name))
147     else:
148         print('Invalid command')
149 
150     print()
151     _save_clients_to_storage()
```

USO DE OBJETOS Y MÓDULOS

34. Decoradores

Python es un lenguaje que acepta diversos paradigmas como programación orientada a objetos y la programación funcional, siendo estos los temas de nuestro siguiente módulo.

Los decoradores son una función que envuelve a otra función para modificar o extender su comportamiento. En Python las funciones son ciudadanos de primera clase, first class citizen, esto significan que las funciones pueden recibir funciones como parámetros y pueden regresar funciones. Los decoradores utilizan este concepto de manera fundamental.

35. Decoradores en Python

En esta clase pondremos en práctica lo aprendido en la clase anterior sobre decoradores.

Por convención la función interna se llama wrapper,

Para usar los decoradores es con el símbolo de @(arroba) y lo colocamos por encima de la función. Es un sugar syntax

*args **kwargs son los argumentos que tienen keywords, es decir que tienen nombre y los argumentos posicionales, los args. Los asteriscos son simplemente una expansión.

36. ¿Qué es la programación orientada a objetos?

La programación orientada a objetos es un paradigma de programación que otorga los medios para estructurar programas de tal manera que las propiedades y comportamientos estén envueltos en objetos individuales.

Para poder entender cómo modelar estos objetos debemos tener claros cuatro principios:

- Encapsulamiento.
- Abstracción
- Herencia
- Polimorfismo

Las clases simplemente nos sirven como un molde para poder generar diferentes instancias.

37. Programación orientada a objetos en Phyton

Para declarar una clase en Python utilizamos la keyword class, después de eso le damos el nombre. Una convención en Python es que todas las clases empiecen con mayúscula y se continua con CamelCase.

Un método fundamental es dunder init(__init__). Lo único que hace es inicializar la clase basado en los parámetros que le damos al momento de construir la clase. self es una referencia a la clase. Es una forma internamente para que podamos acceder a las propiedades y métodos.

38. Scopes and namespaces

En Python, un name, también conocido como identifier, es simplemente una forma de otorgarle un nombre a un objeto. Mediante el nombre, podemos acceder al objeto. Vamos a ver un ejemplo:

```
1  my_var = 5
2
3  id(my_var) # 4561204416
4  id(5) # 4561204416
```

En este caso, el identifier my_var es simplemente una forma de acceder a un objeto en memoria (en este caso el espacio identificado por el número 4561204416). Es importante recordar que un name puede referirse a cualquier tipo de objeto (aún las funciones).

```
1  def echo(value):
2      return value
3
4  a = echo
5
6  a('Billy') # 3
```

Ahora que ya entendimos qué es un name podemos avanzar a los namespaces (espacios de nombres). Para ponerlo en palabras llanas, un namespace es simplemente un conjunto de names. En Python, te puedes imaginar que existe una relación que liga a los nombres definidos con sus respectivos objetos (como un diccionario). Pueden coexistir varios namespaces en un momento dado, pero se encuentran completamente aislados. Por ejemplo, existe un namespace específico que agrupa

todas las variables globales (por eso puedes utilizar varias funciones sin tener que importar los módulos correspondientes) y cada vez que declaramos una módulo o una función, dicho módulo o función tiene asignado otro namespace.

A pesar de existir una multiplicidad de namespaces, no siempre tenemos acceso a todos ellos desde un punto específico en nuestro programa. Es aquí donde el concepto de scope (campo de aplicación) entra en juego.

Scope es la parte del programa en el que podemos tener acceso a un namespace sin necesidad de prefijos. En cualquier momento determinado, el programa tiene acceso a tres scopes:

- El scope dentro de una función (que tiene nombres locales)
- El scope del módulo (que tiene nombres globales)
- El scope raíz (que tiene los built-in names)

Cuando se solicita un objeto, Python busca primero el nombre en el scope local, luego en el global, y por último, en la raíz. Cuando anidamos una función dentro de otra función, su scope también queda anidado dentro del scope de la función padre.

```
1  def outer_function(some_local_name):
2      def inner_function(other_local_name):
3          # Tiene acceso a la built-in function print y al nombre local some_local_name
4          print(some_local_name)
5
6          # También tiene acceso a su scope local
7          print(other_local_name)
```

Para poder manipular una variable que se encuentra fuera del scope local podemos utilizar los keywords global y nonlocal.

```
1  some_var_in_other_scope = 10
2
3  def some_function():
4      global some_var_in_other_scope
5
6      Some_var_in_other_scope += 1
```

39. Introducción a Click

Click es un pequeño framework que nos permite crear aplicaciones de Línea de comandos. Tiene cuatro decoradores básicos:

- @click_group: Agrupa una serie de comandos
- @click_command: Acá definiremos todos los comandos de nuestra apliacion
- @click_argument: Son parámetros necesarios
- @click_option: Son parámetros opcionales

Click también realiza las conversiones de tipo por nosotros. Esta basado muy fuerte en decoradores.

- pip install click : Ver vídeo de instalación: https://www.youtube.com/watch?v=j2Hg56guD4A

40. Definición a la API pública

En esta clase definiremos la estructura de nuestro proyecto PlatziVentas, los comandos, la configuración en nuestro setup.py y la instalaremos en nuestro entorno virtual con pip.

- Ver recursos: https://github.com/ProfeAlbeiro/adso_logica/tree/main/appwebinv2_logica_programacion/appwebinv4_python/proy02_platzi_practical_python_course_CRUD/docs/material/curso_Python3-14-what-is-oop

mkidr : Crea carpetas
touch : Crea archivos
tree . : Ver estructura de carpetas y archivos
pip install virtualenv : Instalar el entorno virtual
python -m venv venv : Crear el entorno virtual
.\venv\Scripts\activate : Activar el entorno virtual

- Si no funciona este comando hacer lo siguiente:
  - Buscar Window PowerShell / clic derecho / Ejecutar como adiministrador
  - Get-ExecutionPolicy -List / Enter
  - Set-ExecutionPolicy RemoteSigned -Scope CurrentUser / Enter / s
    Más información: https://www.cdmon.com/es/blog/la-ejecucion-de-scripts-esta-deshabilitada-en-este-sistema-te-contamos-como-actuar

deactivate : Desactivar el entorno virtual
alias avenv=.\venv\Scripts\activate : Crear un Alias (No funciona aún)
pip install --editable . : Instalar nuestra aplicación

which pv : Línea de Comandos
pv --help : Ayuda General
pv clients --help : Ayuda de la aplicación

41. Clients

Modelaremos a nuestros clientes y servicios usando lo aprendido en clases anteriores sobre programación orientada a objetos y clases.

@staticmethod nos permite declarar métodos estáticos en nuestra clase. Es un método que se puede ejecutar sin necesidad de una instancia de una clase. No hace falta que reciba self como parámetro.

42. Servicios: Lógica de negocio de nuestra aplicación

cat .clients.csv : Visualizar un archivo csv

43. Interface de create: Comunicación entre servicios y el cliente

No hay descripción

44. Actualización de cliente

No hay descripción

45. Interface de actualización

- Ver recursos: https://github.com/ProfeAlbeiro/adso_logica/tree/main/appwebinv2_logica_programacion/appwebinv4_python/proy02_platzi_practical_python_course_CRUD/docs/material/curso_Python3-15-inheritance-polymorphism

46. Manejo de errores y jerarquía de errores en Python

Python tiene una amplia jerarquía de errores que nos da posibilidades para definir errores en casos como donde no se pueda leer un archivo, dividir entre cero o si existen problemas en general en nuestro código Python. El problema con esto es que nuestro programa termina, es diferente a los errores de sintaxis donde nuestro programa nunca inicia.

Para ""aventar"" un error en Python utilizamos la palabra raise. Aunque Python nos ofrece muchos errores es buena práctica definir errores específicos de nuestra aplicación y usar los de Python para extenderlos.

Podemos generar nuestros propios errores creando una clase que extienda de BaseException. Si queremos evitar que termine nuestro programa cuando ocurra un error, debemos tener una estrategia. Debemos utilizar try / except cuando tenemos la posibilidad de que un pedazo de nuestro código falle

- try : Significa que se ejecuta este código. Si es posible, solo ponemos una sola línea de código ahí como buena práctica
- except : Es nuestro manejo del error, es lo que haremos si ocurre el error. Debemos ser específicos con el tipo de error que vamos a atrapar.
- else : Es código que se ejecuta cuando no ocurre ningún error.
- finally : Nos permite obtener un bloque de código que se va a ejecutar sin importar lo que pase.

47. Context managers

Los context managers son objetos de Python que proveen información contextual adicional al bloque de código. Esta información consiste en correr una función (o cualquier callable) cuando se inicia el contexto con el keyword with; al igual que correr otra función cuando el código dentro del bloque with concluye. Por ejemplo:

```
1  with open('some_file.txt') as f:
2      lines = f.readlines()
```

Si estás familiarizado con este patrón, sabes que llamar la función open de esta manera, garantiza que el archivo se cierre con posterioridad. Esto disminuye la cantidad de información que el programador debe manejar directamente y facilita la lectura del código. Existen dos formas de implementar un context manager: con una clase o con un generador. Vamos a implementar la funcionalidad anterior para ilustrar el punto:

```
1  class CustomOpen(object):
2      def __init__(self, filename):
3          self.file = open(filename)
```

```
4
5     def __enter__(self):
6         return self.file
7
8     def __exit__(self, ctx_type, ctx_value, ctx_traceback):
9         self.file.close()
10
11  with CustomOpen('file') as f:
12      contents = f.read()
```

Esta es simplemente una clase de Python con dos métodos adicionales: enter y exit. Estos métodos son utilizados por el keyword with para determinar las acciones de inicialización, entrada y salida del contexto. El mismo código puede implementarse utilizando el módulo contextlib que forma parte de la librería estándar de Python.

```
1   from contextlib import contextmanager
2
3   @contextmanager
4   def custom_open(filename):
5       f = open(filename)
6       try:
7           yield f
8       finally:
9           f.close()
10
11  with custom_open('file') as f:
12      contents = f.read()
```

El código anterior funciona exactamente igual que cuando lo escribimos con una clase. La diferencia es que el código se ejecuta al inicializarse el contexto y retorna el control cuando el keyword yield regresa un valor. Una vez que termina el bloque with, el context manager toma de nueva cuenta el control y ejecuta el código de limpieza.

PYTHON EN EL MUNDO REAL

48. Aplicaciones de Python en el mundo real

Python tiene muchas aplicaciones. En las ciencias tiene muchas librerías que puedes utilizar como analisis de las estrellas y astrofisica; si te interesa la medicina puedes utilizar Tomopy para analizar tomografías. También están las librerías más fuertes para la ciencia de datos numpy, Pandas y Matplotlib.

En CLI por si te gusta trabajar en la nube y con datacenters, para sincronizar miles de computadoras:

- aws
- gcloud
- rebound
- geeknote

Aplicaciones Web:

- Django
- Flask
- Bottle
- Chalice
- Webapp2
- Gunicorn
- Tornado

49. Python 2 vs 3 (Conclusiones)

No es recomendable empezar con Python 2 porque tiene fecha de vencimiento para el próximo año.

PEP = Python Enhancement Proposals. Los PEP son la forma en la que se define como avanza el lenguaje. Existen tres PEPs que debes saber.

- PEP8 es la guía de estilo de cómo escribir programas de Python. Es importante escribir de manera similiar para que nuestro software sea legible para el resto de la comunidad
- PEP257 nos explica cómo generar buena documentación en nuestro código
- PEP20

```
1  import this
2
3  The Zen of Python, by Tim Peters
4
5  Beautiful is better than ugly.
6  Explicit is better than implicit.
7  Simple is better than complex.
8  Complex is better than complicated.
9  Flat is better than nested.
10 Sparse is better than dense.
11 Readability counts.
12 Special cases aren't special enough to break the rules.
13 Although practicality beats purity.
14 Errors should never pass silently.
15 Unless explicitly silenced.
16 In the face of ambiguity, refuse the temptation to guess.
17 There should be one-- and preferably only one --obvious way to do it.
18 Although that way may not be obvious at first unless you're Dutch.
19 Now is better than never.
20 Although never is often better than *right* now.
21 If the implementation is hard to explain, it's a bad idea.
22 If the implementation is easy to explain, it may be a good idea.
23 Namespaces are one honking great idea -- let's do more of those!
```

```
1  Lo bonito es mejor que lo feo.
2  Lo explícito es mejor que lo implícito.
3  Lo simple es mejor que lo complejo.
4  Lo complejo es mejor que lo complicado.
5  Plano es mejor que anidado.
6  Esparcido es mejor que denso.
7  La legibilidad cuenta.
8  Los casos especiales no son tan especiales como para romper las reglas.
9  Aunque la practicidad gana a la pureza.
10 Los errores nunca deben pasar en silencio.
11 A menos que se silencien explícitamente.
12 Ante la ambigüedad, rechaza la tentación de adivinar.
13 Debe haber una -y preferiblemente sólo una- forma obvia de hacerlo.
14 Aunque esa manera puede no ser obvia al principio, a menos que seas holandés.
15 Ahora es mejor que nunca.
16 Aunque a menudo "nunca" es mejor que "ahora mismo".
17 Si la implementación es difícil de explicar, es una mala idea.
18 Si la implementación es fácil de explicar, puede ser una buena idea.
19 Los espacios de nombres son una gran idea: ¡hagamos más!
```

50. Entorno virtual en Python y su importancia

Paquetes de terceros:

- PyPi (Python package index) es un repositorio de paquetes de terceros que se pueden utilizar en proyectos de Python.
- Para instalar un paquete, es necesario utilizar la herramienta pip.
- La forma de instalar un paquete es ejecutando el comando pip install paquete.
- También se puede agrupar la instalación de varios paquetes a la vez con el archivo requirements.txt
- Es una buena práctica crear un ambiente virtual por cada proyecto de Python en el que se trabaje.

Ambientes virtuales:

- Buscar en Google: pip intallation / Clic a la versión más reciente / clic a get-pip.py para ver el script / descargar el script al proyecto / ejecutarlo
- Esto evita conflictos de paquetes en el intérprete principal.
- pip install virtualenv
- virtualenv venv
- .\venv\Scripts\activate
- pip freeze
- pip install flask
- touch requirements.txt
- ls
- requirements.txt / Flask==3.0.0
- pip insatall -r requirements.txt
- deactivate