Python:
Conceptos de Python en Producción

INTRODUCCIÓN A PYTHON

Esta clase

Conocer más del curso y un repaso paso ligero de conceptos

Propósito

Entender los beneficios de Python como un lenguaje multipropósito. Cobertura de conceptos más avanzados para el flujo de desarrollo de aplicaciones en producción así como el empaquetado de proyectos.

Requisitos

1. Conocimientos y noción de Python en cualquier versión
2. Entregables en Github (más información al final de la clase).
3. Atender las clases teóricas

Encuesta

¿Ha usado Github?

La encuesta se abrirá en Zoom

Objetivo

Entender las partes más importantes del lenguaje, y como podemos utilizarlo de manera más eficiente en el contexto de desarrollo en producción.

Fiabilidad y confiabilidad en la manutención del código.

Procesos para automatizar el flujo de desarrollo y la consistencia.

Entornos consistentes y herramientas esenciales para los proyectos en producción.

Sobre su capacitador

Daniel Ortiz

Graduado en Ciencias de la Computación en la Universidad de Toronto (2018)

Maestría en Computación Avanzada, Universidad de Londres – Birkbeck (2021)

Ingeniero de Software, radicado en Londres.

Intereses en aplicaciones e infraestructura de datos. Actualmente en plataformas de generación de datos

En redes sociales 👉

https://linkedin.com/in/danoc93

Estructura del curso

Componente guiado y componente autónomo

- 6 clases vía Zoom, 3 horas por día, Lunes-Viernes 6 – 9 pm desde el 10 de Septiembre.
- Break 10 mins cada hora.
- Entregable escrito (rúbrica a mediados de semana).
- **Diseñado para no afectar su proyecto final y complementar los conceptos.**

https://docs.google.com/document/d/10NQpTOgvEWVPIZPUeiWam-Tj3xgTmVtSZ5q_FVIGEMk/edit

Información

- Repositorio de Github actualizado con los recursos de la clase anterior

- Grupo de Whatsapp para comunicación

Python

Lenguaje multi propósito disponible desde los 80, **muchas versiones con el pasar de los años.**

Python

Mini encuesta (responder en la encuesta del Zoom).

Python 3 vs 2

Python 2 aún es funcional, pero

- Vida útil terminada desde 2020. **No existe soporte comunitario oficial.**
- No existen parches de seguridad oficiales.
- Sintaxis menos flexible.
- Soporte de librerías reducido, limitaciones de código moderno.
- En la práctica muchos proyectos que usan Python 2 se consideran **legado.**

Legado

Código que ha sido heredado de versiones anteriores de un software o sistema.

Este código puede ser difícil de mantener o actualizar debido a su antigüedad, falta de documentación, o porque fue escrito en un lenguaje de programación o estilo que ya no se utiliza comúnmente.

Python 3 vs 2

En Python 3 se mejoró mucho la sintaxis al convertir a todas las operaciones en funciones y **el costo de ejecutar las operaciones .**

Desde el 2023, la versión es 3.12 y la terminología de este curso se enfocará en ésta.

Python 3.12 https://www.python.org/downloads/release/python-3120/

Python 3.12

A pesar de tener esta presentación, el tutorial oficial de Python en español se puede encontrar aquí

El tutorial de Python — documentación de Python – 3.12

https://docs.python.org/es/3/tutorial/

Recomendado depender de la documentación oficial cuando sea posible

Preferir Python 3

Recomendaciones generales:

- Para proyectos nuevos, explorar usar la última versión que cumpla los requisitos.
- Evitar versiones que han vivido su vida útil por falta de soporte y funcionalidad.
- En práctica: **Dedicar tiempo al pago de la deuda técnica.**

Deuda técnica

Deuda técnica es el costo futuro de trabajo adicional causado por elegir una solución más rápida y fácil ahora en lugar de una mejor y más laboriosa.

Ganancias a corto plazo vs sostenibilidad a largo plazo en el desarrollo de software.

Particularmente importante en proyectos grandes o a nivel empresarial.

¿Por qué usar Python?

¡Un lenguaje gratuito y de código libre! Todos pueden colaborar en el lenguaje y revisar como funciona sin ninguna obligación financiera.

Es multiplataforma, es decir que podemos ejecutarlo en PC, Mac, o inclusive desde teléfonos móviles que tengan capacidades más avanzadas.

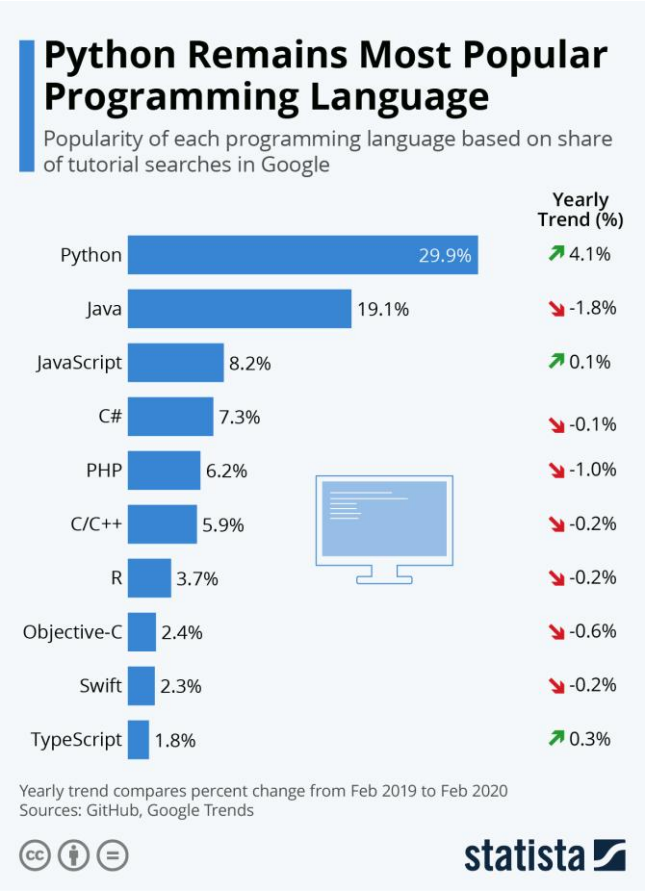
¿Por qué usar Python?

Python es un lenguaje muy fácil de usar como veremos, tiene una muy amplia cantidad de librerías gratuitas que permiten crear proyectos de muchos tipos.

Aplicaciones web, Análisis de datos, Machine Learning, Automatización: Todo esto es posible mediante una serie de ecosistemas libres disponibles y usados por cientos de miles de usuarios.

¡Usado por grandes y pequeñas compañías!

¿Por qué usar Python?



¿Por qué usar Python?

Al ser un lenguaje muy legible, la curva de aprendizaje es muy corta. ¡Se puede empezar a trabajar con él una vez instalado en cuestión de minutos!

Java

```java
File dir = new File("."); // get current directory
File fin = new File(
    dir.getCanonicalPath() + File.separator + "Code.txt"
);

FileInputStream fis = new FileInputStream(fin);

// Construct the BufferedReader object
BufferedReader in = new BufferedReader(new InputStreamReader(fis));

String aLine = null;
while ((aLine = in.readLine()) != null) {
    // //Process each line, here we count empty lines
    if (aLine.trim().length() == 0) {}
}

// do not forget to close the buffer reader
in.close();
```

Python

```python
my_file = open("/home/xiaoran/Desktop/test.txt")

print(my_file.read())
my_file.close()
```

¿Por qué usar Python?

Muchos paradigmas, en Python podemos escribir aplicaciones orientadas a objectos, funcionales o imperativa.

En su momento veremos en más detalle como operar estos modelos con el lenguaje.

¿Cómo funciona Python?

Python es un lenguaje dinámico interpretado.

¿Interpretado? Significa que existe un intérprete, un programa encargado de procesar el código de Python en tiempo real en lugar de compilarlo desde el principio.

Tiene sus ventajas ya que permite colaborar en el mismo código sin necesidad de preocuparse de la plataforma.

Tipado en Python

Naturalmente el lenguaje no tiene tipos al ser dinámico.

En lenguajes estáticos:

```
int num = 5;

num = ´texto´

# ERROR!!!!!!!!!!!!!!!
```

En Python

```
num = 5

num = ´texto´

# No hay problemas
```

Tipado en Python

Esto tiene sus beneficios como desventajas, en siguientes clases aprenderemos como agregar tipado a Python en caso de ser deseado. Esto tiene beneficios especiales como por ejemplo asegurar que el código sea consistente.

Como veremos en otra clase, existen conceptos de tipado en Python a través de funciones modernas del lenguaje que se pueden usar de manera opcional.

Concepto: Programas

¿Qué es un programa?

Instrucciones definidas para que un computador las ejecute: Una calculadora, un procesador de texto, inclusive una página web compleja cuenta como un programa.

Concepto: Abstracción

¿Alto nivel, bajo nivel?

El nivel de abstracción de un lenguaje definen que tan accesible y legible es para un humano entender y trabajar él.

Python es un lenguaje de nivel alto.

Python: Lenguaje de alto nivel

Mientras más bajo el nivel, el programa se asemeja más a las instrucciones directas al computador, en la arquitectura en la que se ejecuta.

-> Código de máquina 1012101

-> Ensamblador (Instrucciones específicas en la arquitectura)

-> C (Control directo de memoria, pocas librerías, gestión de casi todo)

-> Java (Menos control, gestión automática de memoria)

-> Python , Javascript, etc. (Interfaces muy restringidas, no se puede acceder o es necesario gestionar recursos de memoria, mucho más legibles)

Python: Lenguaje de alto nivel

Beneficios

- Portabilidad y multiplataforma

- Bueno para proyectos y aplicaciones que requieran poco trabajo ingenieril

Desventajas

Eficiencia de operaciones que requieren control detallado de la memoria y recursos (por ejemplo aplicaciones con muy baja latencia)

Python: Lenguaje interpretado

Compilación es el proceso de traducir instrucciones del código de fuente en instrucciones que se entiendan por la máquina en la que se ejecutan.

Normalmente sucede una sola vez, y produce binarios que son ejecutables solo en la arquitectura en la que se compiló.

Python: Lenguaje interpretado

Beneficios de los lenguajes compilados

- El proceso de compilación sucede una vez, el programa en sí se puede ejecutar sin necesitar dependencias.
- Si se conocen las arquitecturas de interés, se pueden preparar binarios para todas.

Problemas de la compilación

- Toma tiempo y recursos
- Limitada a la plataforma específica, no hay portabilidad

Python: Lenguaje interpretado

Interpretación es el proceso en el que las instrucciones en lugar de traducirse a lenguaje de máquina una sola vez, se traducen al momento de su ejecución mediante un programa conocido como el intérprete.

El intérprete fue compilado para las arquitecturas de interés, pero cualquier código que quiera ejecutarse a través de él puede ser ejecutado en cualquier momento y cualquier plataforma.

Python: Lenguaje interpretado

Beneficios de los lenguajes interpretados

- Al no haber compilación, el código es portable y puede usarse en muchas plataformas.
- Pocas dependencias de sistema, solo se necesita el intérprete.
- Tipado dinámico.

Problemas de la interpretación

- Al necesitar traducir las instrucciones en tiempo real, existe un costo.
- Limitado al intérprete, es decir, código en Python 3.12 no se puede ejecutar en un intérprete anterior.

Concepto: Paradigmas

Existen distintos paradigmas de programación

En términos simples esto significa, maneras de estructurar programas y trabajar con instrucciones.

Distintos lenguajes soportan diferentes mecanismos, los más comunes son procedural, orientado a objetos y funcional.

Python: Multiparadigma

Paradigma Procedural

Los programas declaran funciones o rutinas (“procedimientos”).

El código se estructura para depender de llamadas a estas funciones.

Esencialmente una lista de instrucciones lineales.

```
def multiplicar(numero1, numero2):
    return numero1*numero2

a = 1
b = 2
print(multiplicar(a, b))
```

Python: Multiparadigma

Paradigma Orientado a Objetos

Se basa en el concepto de objetos, que pueden contener data en diferentes atributos, y código para operar en estos atributos, a manera de procedimientos conocidos como métodos.

Dependiendo del lenguaje, los objetos pueden definirse mediante clases como en Java o Python, o mediante prototipos como en JavaScript.

```python
class Auto:
    def __init__(self, pasajeros):
        self.pasajeros = pasajeros

    def obtener_pasajeros(self):
        return self.pasajeros

auto1 = Auto(5)
print(auto1.obtener_pasajeros())
```

Python: Multiparadigma

Paradigma Funcional

Se basa en el concepto de que computar es la ejecución de funciones matemáticas que mantiene un estado inmutable, es decir una secuencia de funciones que garantizan que nada fuera de ellas ha cambiado durante toda la ejecución.

Un mundo diferente, recomendado aprenderlo, pero no se cubrirá en este curso.

Python ofrece ciertas facilidades para programación funcional pero existen lenguajes como Haskell enteramente dedicados a ella.

Características

- Lenguaje sencillo de entender
- Interpretado y dinámico
- Ofrece estructuras de datos de alto nivel que permiten ser productivo de manera rápida, por ejemplo listas, tuplas y diccionarios.
- Multiplataforma y listo para usar.
- Sistema de módulos muy extenso (más de 300000)

Ecosistema muy popular

- Dada la facilidad de desarrollo e integración existen muchas librerías y entornos populares para diferentes campos.
- En este curso estudiaremos algunas de ellas.

Python: Lo esencial

Su biblioteca estándar viene de fábrica con muchas utilidades multipropósito

Servicios de sistema para acceder y gestionar archivos, hora, subprocesos, rutinas para conectarse a la red, herramientas para operar con compresión, expresiones regulares, email, clientes HTTP, de correo, librerías de interfaz gráfica, matemática de números reales, utilidades de criptografía, etc.

La Biblioteca Estándar de Python — documentación de Python – 3.12

Python: Intérprete

Usar el intérprete interactivo es muy útil para experimentar de manera rápida, sin embargo podemos colocar el mismo código en un archivo y ejecutarlo.

Los resultados no varían, pero claramente un archivo permite distribuirlo.

La extensión común para código de fuente en Python es **.py**

Python: Intérprete

Dada la naturaleza del lenguaje, en la que uno puede declarar variables y cambiar su tipo de manera dinámica.

```
>>> a = 6       ## variable
>>> a           ## imprimir su valor
6
>>> a + 2
8
>>> a = 'hola'    ## 'a' cambiamos su tipo
>>> a
'hola'
>>> len(a)      ## con len() podemos determinar el tamaño
4
>>> a + len(a)  ## concatenemos... algo que no funcionaría?
Traceback (most recent call last):
  File "", line 1, in
TypeError: can only concatenate str (not "int") to str
>>> a + str(len(a))  ## cambiamos su tipo para poder concatenar
'hola4'
>>> noexiste         ## algo que no existe
Traceback (most recent call last):
  File "", line 1, in
NameError: name ‘noexiste' is not defined
```

Python: Código de fuente

La extensión común para código de fuente en Python es **.py**

Cada archivo **.py** se conoce como **módulo**

Un módulo puede importar otros módulos tal como veremos después.

La manera más sencilla de ejecutar un módulo es llamando al intérprete

Por ejemplo

> python3.12 mi_modulo.py

Python: Código de fuente

mi_modulo.py

```
# Los módulos se pueden importar así
# sys es un módulo estándar con acceso a funciones del sistema
import sys

# Podemos colocar código en funciones
def mi_funcion():
    # Los argumentos pasados al programa al ejecutarlo se pueden
encontar aquí
    argumentos = sys.argv
    print('Hola', [1])
    # El argumento 0 es el nombre del script, así que se puede ignorar

mi_funcion()
```

Python: Código de fuente

mi_modulo.py

```
# Los módulos se pueden importar así
# sys es un módulo estándar con acceso a funciones del sistema
import sys

# Podemos colocar código en funciones
def mi_funcion():
    # Los argumentos pasados al programa al ejecutarlo se pueden
encontar aquí
    argumentos = sys.argv
    print('Hola', argumentos[1])
    # El argumento 0 es el nombre del script, así que se puede ignorar

mi_funcion()
```

```
$python mi_modulo.py Daniel
Hola Daniel
```

Python: Variables

A diferencia de un programa en otros lenguajes, las variables en Python no requieren declarar un tipo, ni se requiere que el mismo se mantenga durante la ejecución del programa.

```
variable = 1
print(variable*2)
variable = "Ahora soy texto"
print(variable + " y más texto")
```

Importante: Los tipos existen, pero no son especificados. Son inferidos al momento de aplicar operaciones en las variables.

Recomendación

Mantener consistencia de tipos a pesar de que el lenguaje es flexible.

En aplicaciones más grandes, la consistencia es esencial para evitar problemas de lógica.

Python: Funciones

Algo muy importante a notar, es que Python no usa llaves.

Los bloques de código se determinan de acuerdo a su nivel de indentación o espaciado.

Las funciones reciben cualquier número de argumentos.

```python
# Define una función repetir que recibe dos argumentos
def repetir(texto, exclamar):
    """
    Repite el texto 3 veces y si exclamar es verdadero,
    se agrega un símbolo de exclamación.
    """

    # En Python concatenar se puede hacer de muchas formas
    resultado = texto + texto + texto

    if exclamar:
        resultado = resultado + '!!!'
    return resultado

# Imprimir el resultado
print(repetir("Hola", True))

# O tambien colocarlo en una variable
mivariable = repetir("Hola", False)
print(mivariable)
```

Python: Orden de ejecución

Dado a que el código se ejecuta de arriba hacia abajo, las funciones que quieren ser usadas deben haber sido declaradas antes del punto en el que se necesiten, eso aplica tanto para el mismo archivo como para módulos importados.

```
def funcion1():
  print("Hola")

funcion1() #Imprime Hola
funcion2() #Error!!!!!!!

def funcion2():
  print("Adios")
```

```
def funcion1():
  print("Hola")

def funcion2():
  print("Adios")

funcion1() #Imprime Hola
funcion2() #Imprime Adios
```

Python: Indentación importa

Como vimos antes, la indentación es importante ya que determina los bloques de código lógicos.

Un bloque lógico es una clase, una función, una función dentro de una función, un ciclo for o while, etc.

```
def funcion1():
print("Hola")
```

```
def funcion1():
  print("Hola")
```

```
def ejemplo():
  # Notemos como esto pertenece a la función ejemplo
  for elemento in [1, 2, 3]:
    # Y esto pertenece al bucle for
    print(elemento)
  print("Adios")
```

Python: Encuesta

¿Cuál es el número correcto de bloques lógicos en este código?

```
class Clase:
    def decir_hola():
        print("Hola")
        for nombre in ["Daniel", "Maria"]:
            print(nombre)

def otrafuncion():
    c = Clase()
    print(c.decir_hola())
```

La encuesta se abrirá en Zoom

Python: Encuesta

5

```
class Clase:
    def decir_hola():
        print("Hola")
        for nombre in ["Daniel", "Maria"]:
            print(nombre)

def otrafuncion():
    c = Clase()
    print(c.decir_hola())
```

Python: Chequeos

Python chequea muy poco cuando compila el código en el intérprete. Por ejemplo, formato del archivo.

Sin embargo, la gran mayoría de errores se detectan al momento de la ejecución de la línea problemática. En el ejemplo siguiente, a pesar de haber un error, nuestro programa no fallará.

```
a = 1
if a == 5:
    funcionquenoexiste()
print("hola")
```

Python: Indentación importa

¿Cuál es la manera correcta de indentar?

Mientras el número sea consistente, se recomienda usar 4 espacios por bloque, o una tabulación. Sin embargo, también pueden ser 2 dependiendo de las preferencias.

La guía oficial de estilo de Python recomienda 4 espacios por nivel

PEP 8 - Guía de estilos en Python - El Pythonista

Python: Nombres de variables

Mientras no lleven espacios las variables pueden llamarse de muchas formas, es común usar el patrón **mi_nombre_de_variable** con _ como delimitador de palabras.

Existen palabras reservadas como **if**, **while** o **for** que no pueden ser nombres de variables por obvias razones.

PEP8

Guía de estilo comunitaria. Recomendación de los creadores del lenguaje.

Estilo consistente y mantenido por un comité formado por cientos de contribuyentes al lenguaje.

https://peps.python.org/pep-0008/

Recomendación

Usar siempre el patrón de lenguaje de la comunidad, *por consistencia y sostenibilidad* de los proyectos.

Python: Cadenas de texto

Cadenas de texto o strings en Python son grupos de caracteres definidos por un tipo de fábrica llamado **str** que viene con el lenguaje.

En Python podemos usar tanto comillas dobles como simples para definir una cadena de texto.

```
a = "texto"
a = 'texto2'
```

Las cadenas son inmutables, es decir no se pueden cambiar una vez creadas.

Los caracteres individuales se pueden acceder con la sntaxis de [x] donde x es el índice.

```
a = "texto"
b = a[1] #e
```

Python: Cadenas de texto

Las cadenas permiten operaciones básicas para manipulación

```
nombre = "daniel"
print(nombre[2]) #el valor en índice 2
print(len(nombre)) #6
print(nombre + nombre) #danieldaniel
```

Python: Cadenas de texto

Pero también operaciones más complejas

texto.lower(): Convierte la variable texto en minúscula

texto.upper(): Convierte la variable texto en mayúscula

texto.startswith(otrotexto): Determina si la cadena empieza con la otra cadena (e.g. daniel empieza con dan)

Y muchas más…

http://docs.python.org/library/stdtypes.html#string-methods

Python: Encuesta

¿Qué asignará el siguiente código?

```
nombre = "daniel"
nombre[2] = "x"
```

La encuesta se abrirá en Zoom

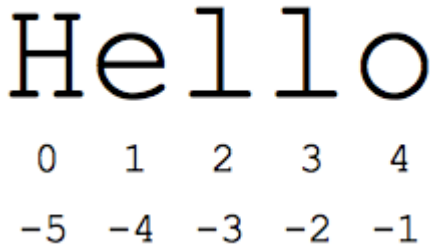
Python: Encuesta

ERROR!

```
Python 3.12 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC
v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> nombre = "daniel"
>>> nombre[2] = "x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python: Cadenas de texto

Uno puede igualmente crear tajadas de cadenas, es decir, obtener subpartes de una cadena, esto también aplicará a listas y estructuras similares.



- s[1:4] es 'ell' – empezando desde 1 hasta, pero sin incluir 4
- s[1:] es 'ello' – omitir algún indice assume el valor máximo, o 0 dependiendo del que se omita.
- s[:] es 'Hello' – omitir ambos índices simplemente clona la cadena porque assume 0 y la longitud total.
- s[1:100] es 'ello' – un índice muy grande se reduce al tamaño máximo

- Los índices negativos se pueden igualmente usar de manera reversible, por ejemplo s[-1] es el último caracter.

Python: Listas y tuplas

Son contenedores de objetos de tamaño variable.

Las listas y las tuplas son similares, con la diferencia de **que las tuplas son inmutables**, es decir no se pueden cambiar una vez definidas.

Pueden contener cualquier tipo de objeto, e inclusive más de uno a la vez.
Las listas se definen con [] y las tuplas con ().

```
lista = [1, 2, "tres"]
tupla = (1, 2, "tres")
```

A pesar de que las tuplas son muy útiles, nos enfocaremos un poco más en las listas.

Python: Listas

Al trabajar con listas podemos aplicar operaciones similares a las cadenas de texto vistas anteriormente, por ejemplo podemos extraer subsecciones, o acceder índices específicos.

```
lista = [1, 2, "tres"]
print(len(lista)) #3
print(lista[1:3]) #[2, "Tres"]
```

A diferencia de las tuplas o las cadenas, las listas son mutables y permiten agregar o remover objetos mediante una serie de funciones disponibles.

lista.append(algo): Insertar algo al final

lista.remove(índice): Remover el elemento en el índice provisto

lista.insert(índice, algo): Insertar algo en el índice especificado

lista.sort(): Ordenar la lista

lista.pop(): Extrae el último elemento de la lista, y lo devuelve en caso de que se necesite

5. Data Structures — Python 3.12 documentation

Python: Condicionales

Como vimos anteriormente, aquí no existen las llaves. Los bloques se definen por indentación.

Bucles, funciones y condicionales se definen mediante :

Se pueden agrupar de muchas maneras y usan los mismos operadores que otros lenguajes ==, !=, <, <=, >, >=.

En un condicional if o while, cualquier valor o función puede usarse, siempre y cuando sean Booleanos.

```
a = 1
if a == 5:
    print("Es 5")
else:
    print("No es 5")
```

Python: Ciclos - while

Así como en otros lenguajes, podemos crear ciclos en Python que ejecuten rutinas repetitivas.

El ciclo **while** necesita una condición que evalúe a un Booleano, de similar manera al condicional **if** que vimos previamente.

```
lista = [1,2,3]
# Mientras la lista tenga elementos
while len(lista) > 0:
    # Extrae el último elemento de la lista y lo asigna a la variable
    elemento = lista.pop()
    print(elemento)
```

Python: Ciclos - for

El ciclo **for** a diferencia de otros lenguajes itera sobre los elementos provistos por un iterador. Un iterador puede ser cualquier implementación que contenta elementos que podamos recorrer.

Por ejemplo, las listas, o las cadenas son iteradores naturales.

La palabra **in** se usa para definir el rango de iteración.

```
lista = [1,2,3]
# Esto significa que elemento tomará cada valor en lista
for elemento in lista:
    print(elemento)
```

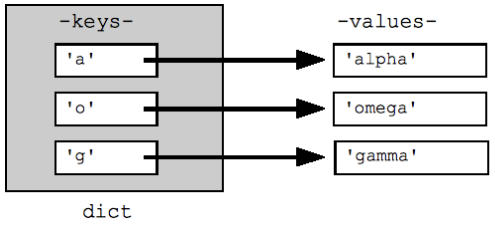
Python: Ciclos - for

En caso de requerir un rango numérico para producir resultados similes a otros lenguajes, se puede usar la función **range**, que genera un iterador para el rango provisto.

```
# Ya que range genera un iterador entre 0 y 99, numero
tendrá todos estos valores.
for numero in range(0, 100):
    print(numero)
```

Python: Diccionarios

Una estructura de datos muy esencial en el lenguaje. Permite definir objectos o mapas entre identificadores y valores. Se define con {}.



Para acceder un valor en un diccionario, podemos usar el identificador asignado y [].

```
cedulas = {
    "Daniel": 1234567,
    "Maria", 0000000
}
cedula_daniel = cedulas["Daniel"]
print(cedula_daniel) #1234567
```

Python: Diccionarios

Al ser muy útiles vienen con una serie de funciones utilitarias que permiten trabajar con ellos más fácilmente.

- diccionario.keys() Iterador de identificadores provistos
- diccionario.values() Iterador de valores correspondientes a los identificadores

Al ser iteradores, podemos imprimirlos con un bucle for:

```
cedulas = {
    "Daniel": 1234567,
    "Maria", 0000000
}
for cedula in cedulas.values():
    print(cedula)
```

Son similares a los objetos en Javascript, pero no exactamente iguales.

Python: Clases

Al igual que otros lenguajes, Python permite definir clases como esqueletos para la instanciación de objetos.

La síntaxis es muy simple.

```python
class Persona:
    # El constructor siempre se define así
    # El argumento self siempre debe estar presente
    def __init__(self, nombre, cedula):
        # Atributos se definen en la instancia self
        self.nombre = nombre
        self.cedula = cedula

    # El argumento self siempre debe estar presente ya que representa al objeto
    def imprimir_datos(self):
        print(f"{self.nombre} tiene cedula {self.cedula}")



# Creamos objetos para esta clase
a = Persona("Daniel", 12134142)
b = Persona("David", 454545)
a.imprimir_datos()
b.imprimir_datos()
```

Laboratorio

Crear un repositorio de Github para los mini laboratorios y enviarlo a

https://forms.gle/WcARAECtj85gdX5KA

Aplicar para beneficios de Github Education (Opcional pero recomendado)
https://education.github.com/discount_requests/application

En el selector buscar Azuay y seleccionar el Instituto Tecnológico del Azuay (Higher Technological Institute of Azuay)

Necesitan cuenta de correo **tecazuay.edu.ec**