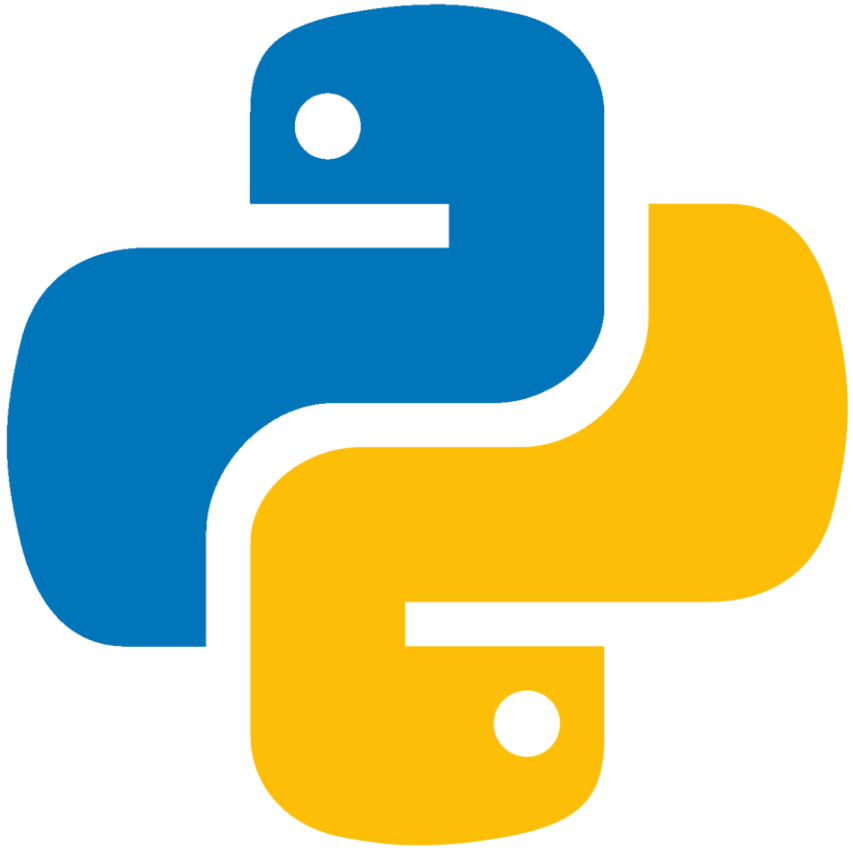


Python: Análisis de Datos y Ecosistemas Modernos de Analítica

INTRODUCCIÓN A PYTHON

Esta clase

CONOCER MÁS DEL CURSO Y UN REPASO PASO LIGERO DE CONCEPTOS

Propuesta

Entender los beneficios de Python como un lenguaje multipropósito.

Cobertura de conceptos y librerías de análisis de datos.

Ecosistema de Python para orquestación de datos a escala.

Requisitos

1. Conocimientos de cualquier lenguaje
2. Cuenta de Github para gestión de material y proyecto
3. Atender las clases teóricas

Objetivos

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

Capacitador

Daniel Ortiz Costa

Ingeniero en Ciencias de la Computación

Universidad de Toronto (2018)

Maestría en Computación Avanzada

Universidad de Londres – Birkbeck (2021)

Ingeniero de Software, enfoque actual en aplicaciones e infraestructura de producción de datos.

Estructura del curso

Componente guiado y componente autónomo

- 1:30 horas por Zoom por 7 días: Agosto 12, 13, 14, 15, 18, 19, 20.
- Clases comienzan a las 6:30 pm.
- Ejercicios recomendados de autoestudio.
- Proyecto en forma de repositorio de Github y reporte

Información

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

https://github.com/danoc93/ista-python-analisis-2025

Usar el grupo de Whatsapp para toda comunicación

¿Herramientas de IA?

Recomendado usar herramientas de inteligencia artificial, pero con **mucho cuidado**.

- ChatGPT/Gemini/Claude son mejores asistentes que maestros
- Siempre verificar información, las alucinaciones son peligrosas.

Python

Mini encuesta anónima para ajustes de contenido (Responder en Zoom)

Python

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

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 heredado de versiones anteriores que sigue en uso, normalmente por su valor o costo de reemplazo.

Puede ser difícil de mantener por antigüedad, falta de documentación o uso de tecnologías obsoletas, y a la vez representar un activo crítico para el sistema.

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

Costo futuro por elegir una solución rápida y fácil en vez de una mejor y más completa.

Implica **ganancias a corto plazo** vs **sostenibilidad a largo plazo**.

Algo crítico en proyectos grandes o empresariales.

¿Por qué usar Python?

Lenguaje libre y sin costo, con código disponible para estudiar, modificar y mejorar.

Multiplataforma: funciona en Windows, macOS, Linux y dispositivos móviles avanzados.

¿Por qué usar Python?

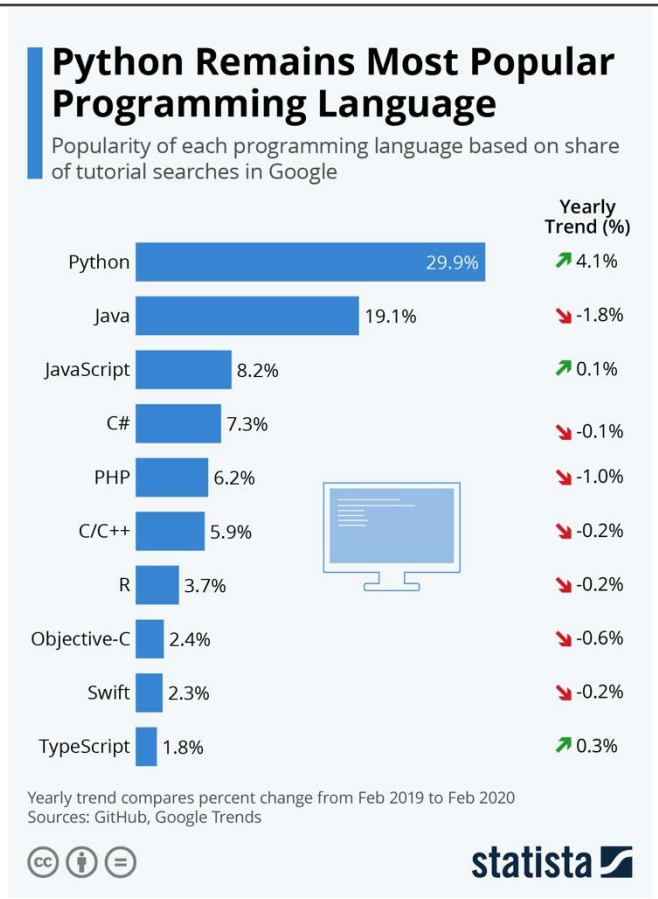
Python: versátil y fácil de usar

Gran cantidad de librerías gratuitas para múltiples áreas.

Aplicaciones web, análisis de datos, machine learning y automatización.

Ecosistema abierto y muy usado en empresas grandes y pequeñas.

¿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?

Interpretado → El código fuente se traduce primero a *bytecode*, que luego es ejecutado por la máquina virtual de Python, en lugar de compilarse a código máquina nativo.

Ventajas:

- Portabilidad: el mismo código funciona en cualquier sistema con un intérprete Python.
- Facilita la colaboración y el intercambio de scripts.
- Desarrollo ágil: ejecución inmediata sin un proceso de compilación largo.

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

Tiene ventajas y desventajas como veremos en otra clase.

En versiones recientes es posible agregar tipado para mejorar la calidad del código.

Beneficio clave: asegura consistencia y reduce errores.

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 o bajo nivel?

El **nivel de abstracción** indica cuán fácil es para un humano leer y trabajar con el lenguaje.

Lenguajes de alto nivel: más legibles, más cercanos al lenguaje humano.

Python es un **lenguaje de alto nivel.**

Python: Lenguaje de alto nivel

Mientras más bajo es el nivel, el programa se acerca más a las instrucciones que entiende directamente el hardware, específicas de su arquitectura. Esto implica **mayor control**, pero también **mayor complejidad**, menos portabilidad y mayor esfuerzo de desarrollo.

Código de máquina – Binario (0s y 1s), nivel más básico, máximo rendimiento, muy difícil de programar.

Ensamblador – Instrucciones por CPU, control total de hardware, eficiente pero difícil y dependiente de arquitectura.

C – Control de memoria, rápido y eficiente, pocas librerías, riesgo de errores de memoria.

Java – Orientado a objetos, multiplataforma (JVM), gestión automática de memoria, menos control.

Python, JavaScript, etc. – Alto nivel, sintaxis simple, memoria abstraída, muy productivos pero poco control.

Python: Lenguaje de alto nivel

Beneficios

Portabilidad y ejecución multiplataforma.

Facilitan el desarrollo y mantenimiento, incluso en proyectos complejos.

Desventajas

Menor control directo sobre memoria y recursos del sistema.

En ciertos casos, pueden ser menos eficientes para tareas que requieren rendimiento extremo o muy baja latencia.

Python: Lenguaje interpretado

La compilación generalmente traduce el código fuente a instrucciones que entiende la máquina, generando un binario específico de la arquitectura.

En Python, esta traducción es dinámica: el código se convierte a *bytecode* y se ejecuta mediante un intérprete, sin generar un binario estático, a diferencia de lenguajes como C o C++.

Compilación estática

Beneficios

El código se compila una sola vez y el programa resultante puede ejecutarse sin dependencias adicionales. Si se conocen las arquitecturas de destino, es posible generar binarios para todas ellas.

Problemas

Requiere tiempo y recursos para compilar.

Binarios limitados a la plataforma específica, sin portabilidad directa.

Python: Lenguaje interpretado

Es el proceso en el que las instrucciones se traducen a lenguaje de máquina **en tiempo de ejecución**, mediante un programa llamado **intérprete.**

El intérprete está compilado para las arquitecturas de destino, pero cualquier código que se ejecute a través de él puede funcionar en cualquier momento y en cualquier plataforma compatible.

Python: Lenguaje interpretado

Beneficios

Portabilidad inmediata: el mismo código puede ejecutarse en distintas plataformas sin recompilación.

Mínimas dependencias: solo requiere el intérprete adecuado.

Desventajas

Menor rendimiento: las instrucciones se traducen en tiempo real.

Dependencia de versión: código para una versión concreta (ej. Python 3.12) puede no funcionar en intérpretes anteriores.

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

Organiza el código en **funciones o procedimientos** que se llaman para realizar tareas específicas. El flujo del programa sigue una **secuencia de instrucciones**, donde cada paso depende de la ejecución de procedimientos previos.

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

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

Python: Multiparadigma

Paradigma Orientado a Objetos

Organiza el código en **objetos** que combinan **estado** (atributos) y **comportamiento** (métodos).
Utiliza principios como **encapsulamiento**, **herencia** y **polimorfismo** para crear software reutilizable.
Según el lenguaje, los objetos se definen con **clases** (Java, Python) o **prototipos** (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

Modelo de programación donde el cálculo se realiza mediante **funciones puras**, evitando modificar el estado y los datos mutables.

Se centra en la **inmutabilidad**, la **composición de funciones** y la ausencia de efectos secundarios. Python ofrece algunas herramientas funcionales, mientras que lenguajes como **Haskell** o **Clojure** están diseñados específicamente para este enfoque.

Características

- **Sintaxis clara y fácil de aprender**
- **Interpretado y de tipado dinámico**
- Incluye **estructuras de datos de alto nivel** como listas, tuplas y diccionarios, que aceleran el desarrollo.
- **Multiplataforma** y listo para usar sin configuración compleja.
- Ecosistema amplio con **cientos de miles de módulos** disponibles (más de 300 000).

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

Usando IA

Herramientas como ChatGPT son muy efectivas para aprender un lenguaje, pero siempre que se pueda respaldar con **fuentes confiables** y, de ser posible, **documentación oficial**.

Prompts buenos (claros, específicos):

"Explica la gestión de memoria en Python vs Java usando la documentación oficial."

"Ejemplo de listas por comprensión en Python siguiendo la guía oficial."

Prompts malos (vagos, sin contexto):

"Explícame Python"

"Hazme un programa"

Python: Intérprete interactivo

El **intérprete interactivo** (REPL) incluido con la instalación de Python es ideal para pruebas rápidas y experimentar.

El mismo código puede guardarse en un archivo **.py** para ejecutarlo, guardarlo y compartirlo.

El resultado es el mismo, pero el archivo facilita la distribución y el uso en proyectos.

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** en Python define los **bloques de código lógicos**, es decir, conjuntos de instrucciones que se ejecutan juntas, como el **módulo entero**, el **cuerpo de una función o clase**, **un ciclo** for o while, o una condición **if/elif/else.**

```
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.

Eventualmente cubriremos herramientas automatizadas para facilitar estos procesos, o inclusive podemos usar herramientas de IA para confirmar patrones utilizados.

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? Índice

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

La encuesta se abrirá en Zoom

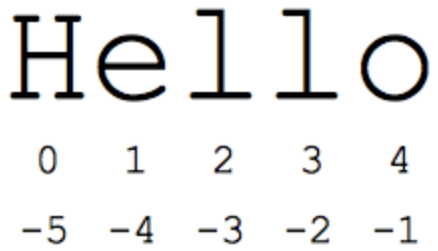
Python: Encuesta

¡ERROR, no podemos asignar valores debido a la inmutabilidad!

```
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 subcadenas, en Python, esto también aplicará a listas y estructuras similares.



La sintaxis **cadena[inicio:fin]** permite realizar estas operaciones de manera sencilla.

Por ejemplo: s[1:4] → 'ell', s[1:] → 'ello', s[:] → copia completa, s[-1] → último carácter.

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)
```

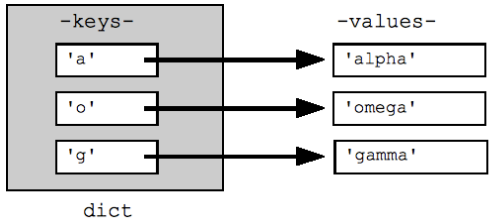
Python: Ciclos - for

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://docs.google.com/forms/d/e/1FAIpQLSdj_q__PLBJk9iXBou3AcjkUO2UCfkvPLiL0O_HuxBmcx4SZQ/viewform?usp=sharing&ouid=113318428653047231876

Aplicar para beneficios de Github Education (Opcional)
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**