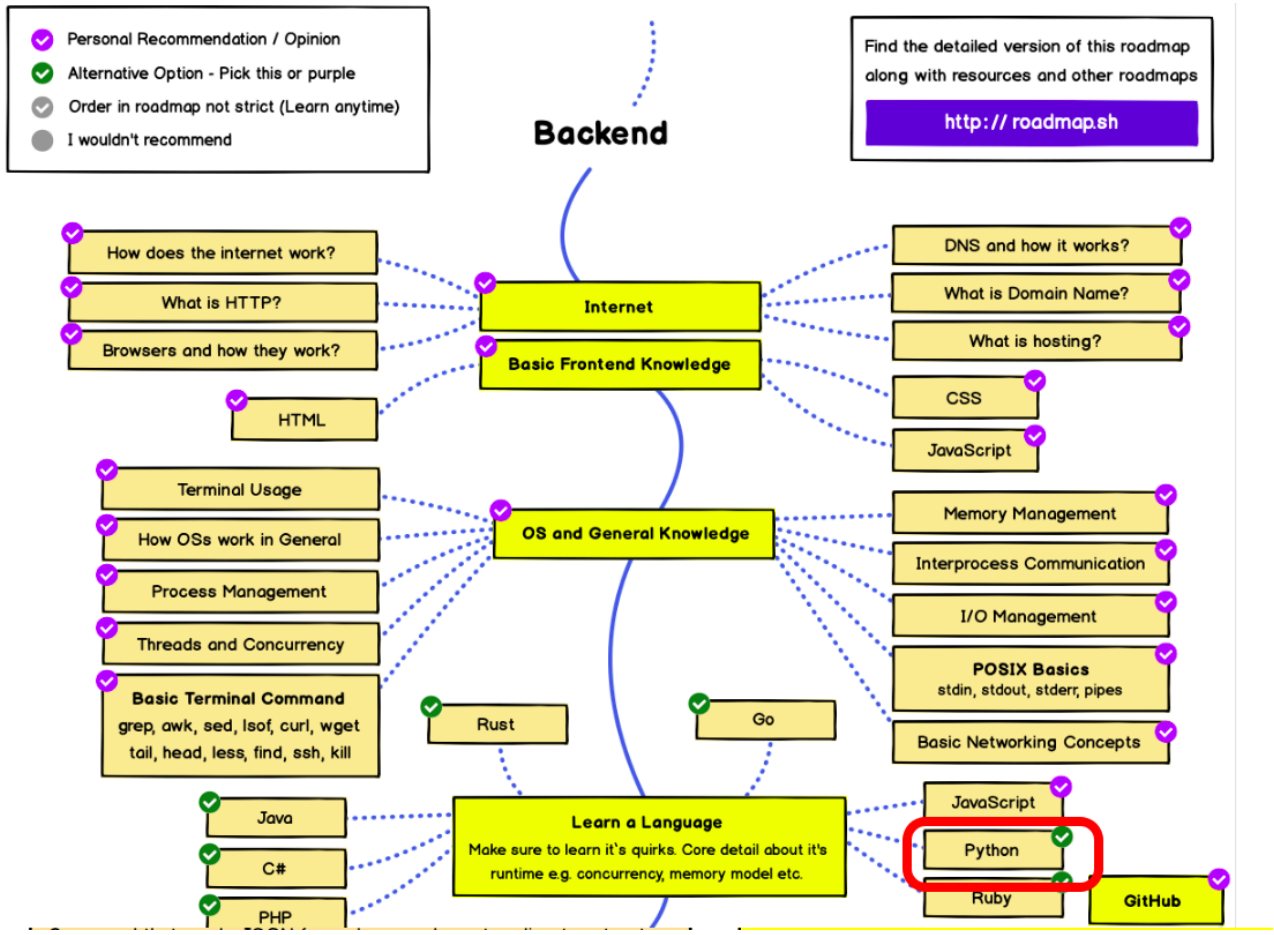
Proyecto de Software

Cursada 2024

¿Qué abordaremos en esta clase?

- Una introducción a Python.
  - Aspectos básicos.
  - POO
- Para más detalle: material del seminario
- También pueden solicitar matricularse en el curso del Seminario 2024.

Programación en el servidor



Roadmap

Nosotros usaremos Python

¿Por qué?

- Es un lenguaje que en los últimos años ha crecido de manera constante.
  - [Stack Overflow Trends

](https://insights.stackoverflow.com/trends?tags=java%2Cc%2Cc%2B%2B%2Cpython%2Cc%23%2Cvb.net%2Cjavascript%2Cassembly%2Cphp%2Cperl%2Cruby%2Cvb%2Cobjective-c) - https://githut.info/

Documentación y referencias

- Sitio oficial: http://python.org/
- Documentación en español: https://wiki.python.org/moin/SpanishLanguage
- Python Argentina: http://python.org.ar/
- Otras referencias:
  - https://docs.python-guide.org/
  - https://realpython.com/

IMPORTANTE: en los tutoriales y cursos en línea chequear la **versión** de Python.

Características del lenguaje

Es un lenguaje de alto nivel, fácil de aprender. Muy expresivo y legible.

In [ ]:
```
random_number = random.randrange(5)
i_won = False
print("Tenés 2 intentos para adivinar un entre 0 y 4")
tries = 1

while tries < 3 and not i_won:
    entered_number = int(input("Ingresá tu número: "))
    if entered_number == random_number:
        print("Ganaste!")
        i_won = True
    else:
        print("Mmmm ... No, ese número no es... Seguí intentando.")
        tries += 1
if not i_won:
    print("Perdiste :(")
    print(f"El número era: {random_number}")
```

- Sintaxis muy clara. **Indentación obligatoria.**

Importante: la legibilidad

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.

- Readability counts.
- ... The Zen of Python

Python Enhancement Proposals (PEP)

- Las PEP son documentos que proporcionan información a la comunidad de Python sobre distintas características del lenguaje, novedades en las distintas versiones, guías de codificación, etc.
- La PEP 0 contiene el indice de todas las PEP.
- La PEP 20: el Zen de Python...

PEP 8: guía de estilo de codificación

"El código es leído muchas más veces de lo que es escrito" (Guido Van Rossum)

- PEP 8
- Hay guías sobre la indentación, convenciones sobre los nombres, etc.
- Algunos IDE chequean que se respeten estas guías.
- Su adopción es MUY importante cuando se comparte el código.

Características del lenguaje

- Es **interpretado**, **multiplataforma** y **multiparadigma**
- Posee tipado dinámico y fuerte.
- Tiene un manejo eficiente de estructuras de datos de alto nivel.

Sentencia **import**

In [ ]:
```
import string
import random
```

In [ ]:
```
letras = string.ascii_lowercase
num = random.randrange(4)
num
print(random.choice(letras))
```

In [ ]:
```
from math import sqrt
raiz = sqrt(16)
print(raiz)
```

random.choice() vs. sqrt()

¿Por qué no math.sqrt() o choice()?

Algunos detalles básicos

In [ ]:
```
# Se usan triple comillas para cadenas de más de una línea
print("""
    En Argentina nací
    Tierra del Diego y Lionel
    De los pibes de Malvinas
    Que jamás olvidaré.
""")
```

In [ ]:
```
valor = input('Ingresa algo que empiece con "a": ')
type(valor)
```

Los f-String

- Fueron introducidos a partir de la versión 3.6.
- Ver la PEP 498
- +Info en la documentación oficial
- Una forma más sencilla de usar el **método format**:

In [ ]:
```
x = 4
print("{0:2d} - {1:3d} - {2:4d}".format(x, x*x, x*x*x))
```

In [ ]:
```
x = 4
print(f"{x:2} - {x*x:} - {x*x*x:4}")
```

In [ ]:
```
cad1 = "En Argentina nací"
cad2 = "Tierra del Diego y Lionel"
cad3 = "De los pibes de Malvinas"
cad4 = "Que jamás olvidaré."
```

In [ ]:
```
print(f"La mejor canción de todas:\n{cad1:<30}\n{cad2:>50}")
print(f"\n{cad3:^30}")
print(f"\n{cad4:-^50}")
```

Tipos de datos

- Tipos predefinidos: (Built-In Data Types)
    - Números (enteros, flotantes y complejos)
    - Booleanos
    - Cadenas de texto
    - Listas, tuplas, diccionarios y conjuntos.

```
gane = False
texto_1 = 'Adivinaste!'
intento = 1
temperatura = 17.5
```

- ¿Qué nos indica un tipo de datos?

Colecciones básicas

In [ ]:
```
cadena = "The Beatles"
lista = ["John", "Paul", "Ringo", "George"]
```

```
tupla = ("John", "Paul", "Ringo", "George")
diccionario = {cadena: lista, 2: tupla}
```

In [ ]:
```
print(cadena[0])
print(lista[0])
print(tupla[0])
print(diccionario)
```

In [ ]:
```
print(diccionario[cadena])
```

¿Qué podemos decir de estas colecciones?

Listas, tuplas, diccionarios

- Mutables e inmutables

```
cadena = "The Beatles"
lista = ["John", "Paul", "Ringo", "George"]
tupla = ("John", "Paul", "Ringo", "George")
diccionario = {cadena: lista, 2: tupla}
```

- ¿Modificamos estas secuencias?

In [ ]:
```
tupla[0]= "Bruce"
tupla
```

Son todas referencias...

In [ ]:
```
rock = ["Riff", "La Renga", "La Torre"]
blues = ["La Mississippi", "Memphis"]
musica = rock
rock.append("Divididos")

musica
```

Estructuras de control: sentencias condicionales

- if
- if .. else
- if .. elif.. elif.. else
- A if C else B

In [ ]:
```
criptos = ["DAI", "USDT"]
cripto = "DAI"
tipo_cripto = "estable" if cripto in criptos  else "cambiante"
print(f"{cripto} es {tipo_cripto}")
```

IMPORTANTE: Python utiliza la **evaluación con circuito corto**.

In [ ]:
```
x = 1
y = 0
if True or x/y:
    print("Mmmm raro...")
else:
    print("nada")
```

match

PEP 636 --> Para Python 3.10

In [ ]:
```
mes = 2
match mes:
    case 1:
        print("Enero")
    case 2:
         print("Febrero")
    case 3:
        print("Ups... Se acabaron las vacaciones!!! :()")
```

In [ ]:
```
cadena = "uno"
match cadena:
    case "uno":
        print("UNO")
    case "dos"  | "tres":
        print("DOS O TRES")
    case _:
        print("Ups.. ninguno de los anteriores")
```

Estructuras de control: iteraciones

- while
- for .. in

In [ ]:
```
i = 5
while i >0:
    print(i, end="-")
    i -= 1
```

In [ ]:
```
for num in range(2, 15, 2):
    print(num, end="-")
```

In [ ]:
```
dias = ["domingo", "lunes", "martes", "miércoles", "jueves", "viernes", "sábado"]
for d in dias:
    print(d, end="-")
```

Algo muy interesante: comprensión de listas

(**list comprehension**)

In [ ]:
```
import string

letras = string.ascii_uppercase
print(letras)
```

```
codigos = [ord(n) for n in letras]
print(codigos)
```

In [ ]:
```
pares = [x for x in codigos if x % 2 == 0]
print(pares)
```

In [ ]:
```
dicci = dict([(x, x**2) for x in range(5)])
dicci
```

Funciones

In [ ]:
```
import string
import random

def generar_clave(largo_clave):
    clave = ''
    for c in range(largo_clave):
        clave  += random.choice(letras)
    return clave
```

In [ ]:
```
letras = string.ascii_lowercase
letras += string.ascii_uppercase
letras += string.digits

mi_clave = generar_clave(10)
print(mi_clave)
```

Un poco más sobre funciones

In [ ]:
```
def generar_clave(largo_clave, todo_minusculas=True):
    clave = ''
    for c in range(largo_clave):
        clave  += random.choice(letras)
    if todo_minusculas:
        return clave.lower()
    else:
        return clave
```

In [ ]:
```
mi_clave = generar_clave(8)
mi_clave
```

- Las funciones pueden tener valores por defecto.
- Estos parámetros siempre se ubican **al final** de la lista de parámetros.
- Más información en documentación oficial sobre funciones

¿Cuándo se evalúan los valores por defecto en los parámetros?

In [ ]:
```
como_recorro = "invertido"
def muestro_cadena(cadena, orden=como_recorro):
    """ Esta función retorna la cadena en forma invertida"""

    return cadena[::-1] if orden == "invertido" else cadena[:]
```

In [ ]:
```
muestro_cadena("Hola")
```

In [ ]:
```
como_recorro = "normal"
muestro_cadena("Hola")
```

Los valores por defecto de los parámetros se evalúan una única vez cuando se define la función.

Podemos definir funciones con un número variable de parámetros

In [ ]:
```
def imprimo(*args):
    """ Esta función imprime los argumentos y sus tipos"""

    for valor in args:
        print(f"{valor} es de tipo {type(valor)}")
```

In [ ]:
```
imprimo(1)
print("-"*30)
imprimo(2, "hola")
print("-"*30)
imprimo([1,2], "hola", 3.2)
```

Tarea para el hogar:

```
def imprimo_otros_valores(**kwargs):
    """ ..... """

    for clave, valor in kwargs.items():
        print(f"{clave} es {valor}")
```

Python permite agregar sugerencias de tipos: anotaciones

In [ ]:
```
def headline(text: str, align: bool = True) -> str:
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "-")
```

In [ ]:
```
print(headline("python type checking", align="left"))
```

Si bien estas anotaciones están disponibles en tiempo de ejecución a través del atributo __**annotations**__, no se realiza ninguna verificación de tipo en tiempo de ejecución.

In [ ]:
```
headline.__annotations__
```

Las funciones como "objetos de primera clase"

- **¿Qué significa esto?** Pueden ser asignadas a variables, almacenadas en estructuras de datos, pasadas como argumentos a otras funciones e incluso retornadas como valores de otras funciones.

In [ ]:
```
def doble(x):
    return 2*x

f = doble
f(10)
```

In [ ]:
```
secuencia = range(10)
dobles = map(doble, secuencia)
for elem in dobles:
    print(elem, end="-")
```

In [ ]:
```
# Una posible solución
def ordeno1(cadena):
    """ Implementación usando sort"""

    lista = cadena.split()
    lista.sort(key=str.lower)
    #lista.sort()
    return lista
```

In [ ]:
```
print(ordeno1("Hoy puede ser un gran día para empezar con Python. "))
```

Expresiones lambda

```
lambda x: 2*x
```
Es equivalente a:

```
def doble(x):
        return 2*x
```
Más info sobre lambda

In [ ]:
```
f = lambda a, b=1: a*b
f(2,3)
```

Usos comunes

In [ ]:
```
lista  = [1, 2, 3, 4, 5, 6, 7]

dobles = map(lambda x: 2*x, lista)
pares = filter(lambda x: x%2 == 0, lista)

for elem in pares:
    print(elem, end=" - ")
```

Decoradores

- Un decorador es una función que recibe una función como argumento y extiende el comportamiento de esta última función sin modificarla explícitamente.
- Son muy usados en Python

In [ ]:
```
def decorador(funcion):
    def funcion_interna():
        print("Antes de  invocar a la función.")
        funcion()
        print("Después de invocar a la función.")

    return funcion_interna

def decimos_hola():
    print("Hola!")
```

In [ ]:
```
saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

In [ ]:
```
saludo()
```

En Python escribimos:

In [ ]:
```
def decorador(funcion):
    def funcion_interna():
        print("Antes de  invocar a la función.")
        funcion()
        print("Después de invocar a la función.")
    return funcion_interna

@decorador
def decimos_hola():
    print("Hola!")
```

In [ ]:
```
decimos_hola()
```

Es equivalente a:

```
decimos_hola = decorador(decimos_hola)
```

- +Ínfo
- +Info en español

Python usa **try - except** para el manejo de excepciones

```
try:
    sentencias
except excepcion1, excepcion2:
    sentencias
except:
    sentencias
else:
    sentencias
finally:
    sentencias
```

Veamos un ejemplo

In [ ]:
```
soda = {1:"Charly Alberti", 2:"Gustavo Ceratti", 3:"Zeta Bosio"}

try:
    for clave in range(1,4):
        print(f"{clave} - {soda[clave]}")
except (KeyError):
    print("Clave incorrecta.")
else:
    print("No se levantaron excepciones.")
finally:
    print("Esta línea se muestra siempre")
```

Python FINALIZA el bloque que levanta la excepción

Objetos en Python

In [ ]:
```
class Banda():
    """Define la entidad que representa a una banda"""

    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []

    def get_nombre(self):
        return self._nombre

    def agregar_integrante(self, nuevo_integrante):
        self._integrantes.append(nuevo_integrante)
```

In [ ]:
```
soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
print(soda.get_nombre())
```

- __init__ es el **inicializador** del objeto.
- Es una buena práctica definir los docstrings
- Público y privado.

Los símbolos de subrayados

En este artículo se describe el uso de underscores en Python

Algunos métodos especiales

- __str__
- __lt__, __gt__, __le__, __ge__
- __eq__, __ne__

In [ ]:
```
class Banda():
    """    Define la entidad que representa a una banda   """
```

```
    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []

    def get_nombre(self):
        return self._nombre

    def agregar_integrante(self, nuevo_integrante):
        self._integrantes.append(nuevo_integrante)

    def __str__(self):
        return (f"{self._nombre} está integrada por {self._integrantes}")

    def __eq__(self, otro):
        return (self._nombre == otro.get_nombre())
```

In [ ]:
```
soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
soda.agregar_integrante("Zeta")
print(soda)
```

In [ ]:
```
soda = Banda("Soda Stereo")
seru = Banda("Seru Giran")
soda==seru
```

Propiedades

In [ ]:
```
class Demo:
    def __init__(self):
        self._x = 0

    def get_x(self):
        print("estoy en get")
        return self._x

    def set_x(self, value):
        print("estoy en set")
        self._x = value

    x = property(get_x, set_x)
```

In [ ]:
```
obj = Demo()
obj.x = 10
print(obj.x)
```

La función property()

property() crea una propiedad de la clase.

```
property(fget=None, fset=None, fdel=None, doc=None)

    x = property(getx, setx, delx, "x es una propiedad")
```

- Más info: https://docs.python.org/3/library/functions.html?highlight=property#property

Usamos el decorador @property

In [ ]:
```
class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

obj = Demo()
obj.x = 10 # Esto dará error: ¿por qué?
print(obj.x)
```

- No podemos modificar la propiedad x. ¿Por qué?

El ejemplo completo

In [ ]:
```
class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value
```

In [ ]:
```
obj = Demo()
obj.x = 10
print(obj.x)
```

Python tiene herencia múltiple

In [ ]:
```
class Musico:
    def __init__(self, nombre, puesto=None, banda=None):
        self.nombre = nombre
        self.tiene_banda = banda!=None
        self._banda = banda
        self.puesto = puesto

    def info(self):
        if self.tiene_banda:
            print (f"{self.nombre}  integra la banda {self.banda}")
        else:
            print(f"{self.nombre} es solista ")

    @property
    def banda(self):
        if self.tiene_banda:
            return self._banda
        else:
            return "No tiene banda"

    @banda.setter
```

```
    def banda(self, banda):
        self._banda = banda
        self.tiene_banda = banda!=None
```

In [ ]:
```
class Guitarrista(Musico):
    def __init__(self, nombre, banda=None):
        Musico.__init__(self, nombre, "guitarrista", banda)
        self.instrumento = "guitarra acústica"

    def info(self):
        print (f"{self.nombre} toca {self.instrumento}")
```

In [ ]:
```
class Vocalista(Musico):
    def __init__(self, nombre, banda=None):
        Musico.__init__(self, nombre, "vocalista", banda)
        self.tipo_voz = "Barítono"

    def info(self):
        if self.tiene_banda:
            print (f"{self.nombre} CANTA en la banda {self.banda}")
        else:
            print(f"{self.nombre} es solista ")
```

In [ ]:
```
bruce = Vocalista('Bruce Springsteen')
brian = Guitarrista("Brian May", "Queen")
```

In [ ]:
```
bruce.info()
brian.info()
```

In [ ]:
```
class VocalistaYGuitarrista(Guitarrista, Vocalista):
    def __init__(self, nombre, banda=None):
        Vocalista.__init__(self, nombre, banda)
        Guitarrista.__init__(self, nombre, banda)
```

In [ ]:
```
bruce = VocalistaYGuitarrista('Bruce Springsteen')
bruce.info()
```

A tener en cuenta ...

- MRO "Method Resolution Order"
- Por lo tanto, es MUY importante el orden en que se especifican las clases bases.
- Más información en documentación oficial

In [ ]:
```
VocalistaYGuitarrista.__mro__
```

Métodos de clase

- Se utiliza el decorador **@classmethod**.
- Se usa **cls** en vez de **self**. ¿A qué hace referencia este argumento?

In [ ]:
```
class Banda():
    generos = set()

    @classmethod
    def limpio_generos(cls, confirmo=False):
        if confirmo:
```

```
            cls.generos =set()
        else:
            return cls.generos

    def __init__(self, nombre, genero="rock"):
        self._nombre = nombre
        self._genero = genero
        self._integrantes = []
        Banda.generos.add(genero)
```

In [ ]:
```
soda = Banda("Soda Stereo")
```

In [ ]:
```
Banda.limpio_generos()
```

Recursos sobre el Seminario de Python

Sitio público de la materia: https://python-unlp.github.io

Más específicamente:

- Guías de estilo
- Buenas prácticas

Nos vemos en la próxima ...