Práctica 2. Introducción a Python

Introducción a la Práctica

El objetivo de esta práctica es realizar una pequeña introducción al lenguaje de programación Python. Como ya hemos visto en clase, Python es un lenguaje interpretado (con sus pequeños trucos)[1].

Al igual que Java y C++, la librería estandar de Python está extensamente documentada en Python Docs (`https://docs.python.org/3/index.html`). Además, como curiosidad, las mejoras y el estándar del lenguaje están recogidos en los PEPs (Python Enhancement Proposals) que sirven para documentar los cambios que ha ido sufriendo el lenguaje con el tiempo.

En general, el lenguaje tiene varios tipos de datos primitivos (int, float, str) y algunos contenedores de datos que el lenguaje llama colecciones. El lenguaje, además, es orientado a objetos. Por tanto permite definir clases y agrupar lógica y datos en ellas como hemos visto en otras asignaturas. Además, añadir librerias externas en python es muy sencillo, sólo necesitas ejecutar un comando. Por último, por consistencia a la hora de escribir código, Python define su propia guia de estilo, el PEP-8 (`https://peps.python.org/pep-0008/`)

La filosofía general de python se define muy bien en el PEP-20, tambien llamado el Zen de Python:

```
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.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!
```

Objetivo

El objetivo de la práctica es que el estudiante se familiarice con Python, para ello vamos a explorar los principales tipos de datos y las colecciones, además, veremos como importar librerías (en Python se llaman módulos) y cómo configurar el entorno de Visual Studio Code para ejecutar código en Python.

Preparación del entorno

Lo primero que vamos a hacer es preparar VSCode para trabajar en python, para esto sólo necesitamos instalar la extensión de Python (ms-python.python) que incluye soporte completo para el lenguaje, incluyendo el language server (PyLance), y el debugger (debugPy). Una vez preparado el entorno vamos a probarlo, crea un fichero "prueba.py" y añadele el siguiente texto:

```python
print("hola mundo")
```

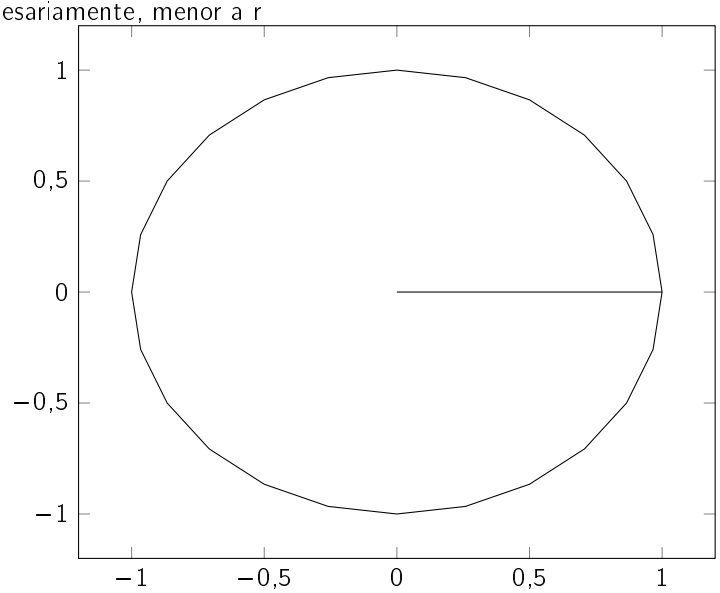
Con esto el entorno quedaría completamente configurado.

[1] `https://peps.python.org/pep-3147/`

Primer ejercicio. Bucles while y primeros pasos

El primer ejercicio de esta práctica se centrará en crear una estimación de π. Para ello vamos a emplear un método basado en Monte Carlo.

Asumamos un circulo, $r = 1$. La distancia de cualquier punto que se encuentre dentro del circulo al centro, debe de ser necesariamente, menor a r



Si lanzamos puntos aleatorios en el rango $[-1, 1]$ podemos comprobar facilmente si están dentro o fuera del circulo calculando si la distancia al punto $(0, 0)$ es menor a 1 ($\sqrt{x^2 + y^2} \leq 1$). Finalmente podemos aproximar $\pi \approx 4\frac{N_{dentro}}{N_{lanzados}}$.

Calcula el valor de pi con un error menor a 10^-2.

Para realizar este ejercicio, vamos a necesitar importar varios modulos de Python, el primero de ellos, el módulo math que contiene funciones útiles relacionadas con las matemáticas. El segundo de ellos el módulo random, que contiene funciones relacionadas con generación de números aleatorios.

Segundo ejercicio. Bucles for

Vamos a crear un generador aleatorio de contraseñas. Para ello vamos a preguntarle al usuario de cuantos carácteres quiere su contraseña, si quiere mayusculas, y signos de puntuación. Por defecto el generador solo utilizará minusculas La solicitud de datos de entrada al usuario se puede hacer con la función input (`https://docs.python.org/3/library/functions.html#input`) y el resultado se puede convertir a un número con la función 'int()'.

Tercer ejercicio. Listas, Funciones

Ayudemos a Papá Noel. Escribe una función que devuelva "naughty" o "nice" dependiendo de las acciones que se le pasen. Las acciones cuya primera letra es una 'b', 'f', o 'k' se consideran "naughty". Las que empiezan con las letras 'g', 's' o 'n' son consideradas nice. Si hay igual número de acciones "naughty" que "nice". La función debe devolver "naughty"

```
bad_actions = ['broke someone\'s window', 'fought over a toaster', 'killed a bug']
what_list_am_i_on(bad_actions) #-> 'naughty'
good_actions = ['got someone a new car', 'saved a man from drowning', 'never got into a fight']
what_list_am_i_on(good_actions) #-> 'nice'
actions = ['broke a vending machine', 'never got into a fight', 'tied someone\'s shoes']
what_list_am_i_on(actions) #-> 'naughty'
```

Cuarto ejercicio, Diccionarios. Funciones II, Funciones como ciudadanos de primera clase

En este ejercicio vamos a implementar un mecanismo de caché. Como sabemos, las funciones en python pueden ser llamadas usando valores, o duos clave-valor. Hay que tener en cuenta que los parámetros de las funciones irán siempre en un order, pero los parametros con nombre, no tienen por que seguir ese order. Por ejemplo:

```
func(a, b, p0=1, p1=2)
func(a, b, p1=2, p0=1)
```

Ejecuta la misma llamada dos veces, aunque el orden de los parámetros no sea necesariamente el mismo.

Por otra parte, en python las funciones son ciudadanos de primera clase. Esto quiere decir que pueden ser tratadas como variables, pueden ser pasadas como parámetro a otra función. Para simplificar el código que se debe de realizar, se va a dar una parte del mismo

```
from collections import OrderedDict

__cache__ = {}

def cache(function, *args, **kwargs):
    kwargs_dict = OrderedDict(sorted(kwargs.items(), key=lambda x: x[0]))
    hash_str = "".join([f"{kw}={arg}:{type(arg)}" for kw, arg in kwargs_dict.items()])
    key = (*args, hash_str)
    ...
```

En el código proporcionado, se ordenan alfabeticamente los parámetros pasados con un nombre a la función, y se crea una tupla con los argumentos y los argumentos nombrados que nos servirá para clave de nuestro diccionario "__cache__" La función caché puede ser utilizada, pruébala.

```
from datetime import datetime
import sys
# Incrementamos el tamaño de la pila
sys.setrecursionlimit(2147483647)


def fibonacci(x):
    if x == 1:
        return 1
    if x == 0:
        return 0
    return fibonacci(x - 1) + fibonacci(x - 2)

def cache_fibonacci(x):
    if x == 1:
        return 1
    if x == 0:
        return 0
    return cache(cache_fibonacci, x - 1) + cache(cache_fibonacci, x - 2)

init = datetime.now()
fibonacci(100)
end = datetime.now()
no_cache_fibonacci_time = end - init

init = datetime.now()
cache_fibonacci(1000)
end = datetime.now()
cache_fibonacci_time = end - init
print(no_cache_fibonacci_time)
print(cache_fibonacci_time)
```