PEP 75.40 -- Guía de estilo y buenas prácticas para Algoritmos y Programación I

Revisiones

Fecha	Versión	Descripción
2020-09-22	0.1	Borrador
2020-10-29	1.0	Primera versión
2021-05-12	2.0	Segunda revisión con comentarios y aportes de Nicolás Marianetti y Diego García Jaime
2021-06-04	2.1	Break (sugerido por Gustavo Bianchi) y Carga de archivos enteros a memoria

PEP:	75.40
Título:	Guía de estilo y buenas prácticas para Algoritmos y Programación I
Autores:	Ezequiel González Busquin <egbusquin at fi.uba.ar>;
Estado:	Activo
Tipo:	Proceso
Creado:	22-Sep-2020
Historial:	22-Sep-2020, TBD

Introducción

La siguiente es una guía de buenas prácticas que se usa como referencia para los ejercicios individuales, grupales y exámenes de la materia Algoritmos y Programación I (Cát. Pablo Guarna) de la FIUBA.

PEP 8

La cátedra considera las buenas prácticas y recomendaciones del PEP 8 como básicas, siempre y cuando no contradigan recomendaciones y prácticas especiales definidas en el presente PEP.

Código original

Los ejercicios, trabajos individuales y los exámenes parciales y finales deben ser de autoría de cada estudiante, no pudiendo ni debiendo cada estudiante compartir su propio código ni reutilizar o adaptar código provisto por algún otro estudiante, ni copiando y pegando código disponible en internet.

Los trabajos prácticos grupales deben ser de autoría de uno, varios o todos (por ejemplo, escritos mediante técnicas de programación de a pares) los miembros del grupo únicamente.

La cátedra utilizará sistemas de detección automática de plagio y tiene el derecho de descalificar al alumno de la cursada, examen, y/o iniciar un sumario en la Facultad según el reglamento.

Versión de Python

La cátedra utiliza el intérprete de **Python 3** para corrección de ejercicios y exámenes, de ser posible en su versión más reciente. Consideraciones especiales deberán acordarse con cada ayudante llegado el caso.

Evitar el uso de Python 2, ya que se encuentra finalizado su ciclo de vida desde 2020.

Para ver la versión de Python instalada, ejecutar el siguiente comando:

```
$ python --version
Python 3.7.5
```

IDE recomendados

Para dar los primeros pasos se recomienda fuertemente el uso de Thonny, ya que permite una ejecución paso a paso muy clara y herramientas para *debugging* de programas.

Otros entornos de desarrollo posibles además de Thonny recomendados por la cátedra son:

- PyCharm
- VS Code con extensión para Python
- Geany

Entornos NO recomendados son todos aquellos que no cuentan con herramientas de *debugging*:

- Sublime
- Notepad++
- Bloc de notas

Números mágicos

Evitar dentro de lo posible el uso de números mágicos (constantes numéricas sin nombre). Se recomienda definir constantes para dejar en claro a qué hace referencia un número dentro de la definición del problema.

Python en realidad no tiene la funcionalidad de hacer que una variable sea constante (y que no pueda ser modificada en cualquier momento), pero el hecho de definirla de esta forma *da a entender* al desarrollador que la intención es que el valor de esa variable no se modifique durante el resto de la ejecución del programa.

```
# Incorrecto:

for i in range(31):
   if ventas[i] > 150:
      alta_demanda.append(i)

# Correcto:

MAX_SUCURSALES = 8
MIN_ALTA_DEMANDA = 150

for i in range(MAX_SUCURSALES):
   if ventas[i] > MIN_ALTA_DEMANDA:
      alta_demanda.append(i)
```

Convención de nombres

Existen diversas convenciones de nombres de identificadores (nombres de variables, funciones, “constantes”) como se vio en el ejemplo anterior. Python recomienda:

`snake_case` para **funciones** y **variables**: Unificar palabras pasando todo a minúscula y uniéndolas por guiones bajos. Se acepta también `camelCase` (concatenar con la primera letra en mayúscula a partir de la segunda palabra), pero se recomienda para módulos que ya usen esa convención.

`MAYÚSCULAS_UNIDAS_POR_GUIÓN_BAJO` para las **constantes** (ver nota en Números mágicos).

`PascalCase` (primera letra de cada palabra en mayúsculas, uniendo sin espacios) para nombres de **clases**.

Lo importante para los ejercicios, trabajos prácticos y exámenes es elegir una convención y mantener la coherencia, evitar mezcla de convenciones. La misma coherencia aplica al nombrar en inglés o español, evitar mezclarlos.

Nombres representativos o mnemónicos

Es recomendado que los nombres de los identificadores de variable, función, constante, clase, campo, etc. sean representativos de la función que cumplen en el código.

Buenos nombres	**Malos nombres**
`temp, temperatura, total, promedio, color, cant_alfajores, nombre_alumno`	`a, t, j, k, p` `v1, v2, v3`

Como regla general, cuanto mayor es el ámbito de uso de una variable, más descriptivo tiene que ser su nombre. Por este motivo se deberían evitar las variables de una única letra, a excepción de una variable de control en un ciclo muy pequeño, donde podría ser aceptable.

Comentarios

Se recomienda agregar comentarios que agreguen valor y aclaren lo que el código hace, sin repetir lo que se entiende claramente al leerlo.

Mal ejemplo de comentario redundante:

```
# Incremento el total de alfajores en uno
cant_alfajores += 1
```

Comentario relevante, explica la lógica detrás del código, algo que no resulta obvio con solo leerlo:

```
# Si la temperatura es elevada agregamos agua para refrescar el motor
if temp > MAX_TEMP_AMBIENTE:
   agua += litros_agua_disponible()
```

Se recomienda también agregar documentación a las funciones con información relevante utilizando la nomenclatura de comentario multilínea:

```
"""
   Función: cargar_aceite_motor
   Parámetros:
```

```
      litros_aceite: número real positivo representando los litros de
                     aceite a cargar
      tiempo: segundos que se tienen disponibles para realizar la carga
   Salidas:
       True si todo resulta bien, False si el aceite rebalsó
   Precondiciones: el motor tiene que estar parado
   Postcondiciones: el aceite que sobra de la carga se descarta
"""
cargar_aceite_motor(litros_aceite, tiempo)
```

Booleanos

Evitar comparar una variable booleana o una condición con `True` o `False`.

```
# Incorrecto:

if es_impar == True:
   # hacer x, y, z

# Correcto:

if es_impar:
   # hacer x, y, z
```

Para comparaciones por `False`, preferir siempre el uso del operador `not`.

```
# Incorrecto:

if es_impar == False:
   # hacer x, y, z

# Correcto:

if not es_impar:
   # hacer x, y, z
```

Comparar explícitamente por `True` o `False` resulta en una tautología, ya que ambas condiciones tienen la misma tabla de verdad.

Valor de la variable `es_impar`	`es_impar`	`es_impar == True`
True	**True**	**True**
False	**False**	**False**

Valor de la variable `es_impar`	`not es_impar`	`es_impar == False`
True	**False**	**False**
False	**True**	**True**

Lo que termina ensuciando el código sin necesidad. Entonces, evitar la comparación siempre que sea posible.

If condensado / Operador ternario

Cuando las condiciones son simples, preferir el uso del if condensado, también conocido como "operador ternario" para abreviar el código sin perder claridad.

```
# No recomendado:

if total > MINIMO and not (codigo_invalido or clave_vencida):
   resultado = total
else:
   resultado = 0
return resultado

# Recomendado:

return total if total > MINIMO and not (codigo_invalido or clave_vencida) else 0
```

Intentar evitar siempre el uso de True o False explícito cuando sea derivado de una condición.

```
# No recomendado, 5 líneas de código:

if x > MAX:
```

```
   resultado = True
else:
   resultado = False

return resultado

# Tampoco (Es equivalente a usar == True o == False):

return True if x > MAX else False

# Recomendado:

return x > MAX
```

Único punto de salida (único `return`)

La cátedra adopta como convención la práctica de tener un único `return` dentro de cada función. Si bien es válido el uso de múltiples return dentro de una función en Python, su práctica puede hacer más difícil de seguir el código o llevar a generar código espagueti.

Dejamos esta práctica libre para que los alumnos la utilicen a criterio fuera de la materia, pero para todos los ejercicios, trabajos prácticos y exámenes se considerará como incorrecto el uso de múltiples `return`.

```
# No recomendado:

def f(x):
   if x < 0:
      return -1
   elif x == 0:
      return 0

   return x ** 2

# Recomendado:

def f(x):
   if x < 0:
      resultado = -1
   elif x == 0:
      resultado = 0
```

```
   else:
      resultado = x ** 2

   return resultado
```

Ejecución normal de bucles sin `break` ni `continue`

Alineado con la política del único punto de salida para facilitar el análisis de los algoritmos y respetar los principios de la programación estructurada es que no haremos uso de sentencias `break` y `continue` para alterar el curso normal de los ciclos `for` y `while` dentro de los programas desarrollados en la cátedra.

```
# No utilizar

for i in range(MAXIMO)
   if i == ALGUN_VALOR:
      break
   print(i)

# Sugerido

i = 0
while (i < MAXIMO) and (i != ALGUN_VALOR):
   print(i)
   i += 1
```

```
# No utilizar

for i in range(MAXIMO)
   if i % 2 == 0:
      print("Ignoramos los números pares...")
      continue
   print(i)

# Sugerido

i = 0
while i < MAXIMO:
```

```
   if i % 2 == 0:
      print(i)
   else:
      print("Ignoramos los números pares...")
   i += 1
```

Cadenas de caracteres con parámetros

Se puede utilizar cualquiera de las formas estándar de imprimir valores en strings.

```
# Estilo printf del lenguaje de programación C
print("Hola, %s!" % nombre)

# str.format
print("Hola, {}!".format(nombre))

# f-strings, disponible desde Python 3.6
print(f"Hola, {nombre}!")

# Plantillas (String Templates), poco usado
from string import Template
t = Template('Hola, $nombre!')
print(t.substitute(nombre=nombre))
```

Intentar mantener coherencia en un mismo programa utilizando el mismo estilo.

Operador `in`

Preferir el uso del operador `in` en una tupla o lista en lugar de comparar una variable contra varios valores uniendo condiciones.

```
# No recomendado:

if dia == LUNES or dia == MARTES or dia == MIERCOLES or dia == JUEVES or
dia == VIERNES:
   # día hábil

# Recomendado:
```

```
if dia in (LUNES, MARTES, MIERCOLES, JUEVES, VIERNES):
   # día hábil
```

Responsabilidad única

Al modularizar un programa, cada función tiene que tener una única responsabilidad. Tener toda la funcionalidad de lectura de teclado, cálculos varios, impresión de resultados, lectura y escritura en archivos, etc. dentro de una misma función dificulta luego la modularización: si necesito hacer el mismo cálculo sin leer de teclado, si no necesito guardar resultados en archivos, ¿puedo reutilizar el código?

No hay una regla dura para esta recomendación del estilo “cada función tiene que tener menos de 10 líneas de código”, pero generalmente una función con **más de 40 o 50 líneas de código probablemente se pueda modularizar** en subfunciones más simples.

Queda a criterio de cada uno encontrar el punto justo de modularización, que se obtendrá con la práctica y aprendiendo de los ejemplos dados en clase.

Niveles de anidamiento

Mantener los niveles de anidamiento del programa (principalmente de sentencias `if-else`) en niveles razonables. Tener muchos niveles de anidamiento (6, 7 o más) hacen que el código se vuelva ilegible y difícil de seguir.

Cuando sea apropiado, preferir usar una sentencia `elif` en lugar de un `if` dentro del bloque `else`.

```
# No recomendado:

if dia == LUNES:
   empezar_semana()
else:
   if hora > 12:
      preparar_almuerzo()
   else:
      if temperatura > 30:
         encender_aire()
      else:
         ...

# Recomendado:
```

```
if dia == LUNES:
   empezar_semana()
elif hora > 12:
   preparar_almuerzo()
elif temperatura > 30:
   encender_aire()
else:
   ...
```

Manejo de errores: Excepciones

Limitar el bloque `try` a la menor cantidad de sentencias posibles. En el ámbito de la materia limitar el uso de excepciones a la validación de entradas (valor numérico válido, rango válido, etc.) y evitar utilizar excepciones para definir la lógica del programa.

Carga de archivos enteros a memoria

Evitar cargar archivos enteros a memoria para procesarlos, ordenarlos, etc. Asumir que cuando se tiene un archivo con datos, estos pueden ser millones de líneas o registros y que su carga total a memoria no resulta posible [complejidad espacial *O(n)*].

Generalmente los ejercicios de procesamiento de archivos se pueden resolver leyendo de a un registro o línea a la vez, procesarlo, escribir las salidas, contabilizar la información, etc. y después leer el próximo hasta llegar al final del archivo usando las técnicas de procesamiento de archivos dados en clase como apareo, mezcla y sus variaciones [complejidad espacial *O(1)*].

En algunos casos es válido cargar a memoria un único dato de cada línea o registro del archivo para su procesamiento. También sería posible cuando el tamaño del archivo se puede estimar limitado, como por ejemplo un archivo que contenga registros con la población de cada Comuna de la Ciudad de Buenos Aires, porque se sabe que las comunas son un número < 20.