Clase 6. Buenas prácticas

TRABAJADORES INFORMÁTICOS
organizados en AGC

Ministerio de Trabajo, Empleo y Seguridad Social
Argentina

¿Qué son las buenas prácticas?

- Son prácticas que son *recomendadas* al momento de programar.
- Tiene carácter de **opcionales** pero se recomienda su aplicación por múltiples motivos, como por ejemplo simplificar el código, mejorar la legibilidad o prevenir bugs.

¿Qué son las buenas prácticas?

- No suelen modificar el comportamiento del programa
- Son prácticas comunes entre todos los desarrolladores del mundo.

Nombres de variables

Utilizar nombres descriptivos, que describan el sentido de la variable. Evitar:

- Tipo de dato en el nombre. Ej: *nombre_string*
- Nombres muy largos
- Nombres que desinforman
- Nombres inentendibles

```
>>> i = 10

>>> array_de_clientes = []

>>> aux = True

>>> cliente = False
```

```
>>> cantidad= 10

>>> clientes = []

>>> control = True

>>> esCliente = False
```

Nombres de variables

Es importante también respetar las **convenciones por lenguaje**.
Por ejemplo en Python se utiliza *underscore* (variables en minúscula usando _ para separación de palabras) en lugar de *camelCase* (sin separación entre palabras, primera letra de cada palabra en mayúscula exceptuando la primera).

```
>>> i = 10

>>> array_de_clientes = []

>>> aux = True

>>> cliente = False
```

```
>>> cantidad= 10

>>> clientes = []

>>> control = True

>>> es_cliente = False
```

Nombres de funciones

Utilizar nombres descriptivos, se entienda su propósito.
Evitar:

- Nombres genéricos.
- Nombres confusos.
- Nombres muy largos.

```
>>> def funcion1(...):
  ...
>>> def se_fija_si_es_impar(...):
  ...
>>> def maximo(...):
  ...
```

```
>>> def ordenar(...):
  ...
>>> def es_impar(...):
  ...
>>> def obtener_maximo(...):
  ...
```

Usar constantes

Un número mágico es un número en el medio del código que depende de la lógica de negocio. Los evitamos porque entendemos su sentido en el momento de agregarlos, pero luego podemos olvidarlo y no vamos a tener idea qué era ese valor.

Para evitarlos usamos constantes, que por convención se escriben en mayúsculas.

```
while (cantidad_clientes < 17):
  ...


total = total + 1.21 * total
```

```
MAXIMO_CLIENTES = 17
while (cantidad_clientes < MAXIMO_CLIENTES):
  ...

IVA = 1.24
total = total + 1.21 * total
```

Mantener coherencia con los estilos

Para facilitar la legibilidad, hay que ser coherentes con las decisiones de estilo de código. Evitar:

- Mezclar idiomas
- Mezclar estándares de nomenclatura:
  - Elegir una y mantenerla.
  - Existen:
    - camelCase
    - PascalCase
    - snake_case (se suele utilizar para Python)
    - kebab-case

```
def es_impar(number):
  variable_con_snake = number
  numeroParaVerPar = 2
  return variable_con_snake % numeroParaVerPar
```

```
def es_impar(numero):
  variable_con_snake = numero
  numero_para_ver_par = 2
  return variable_con_snake %
numero_para_ver_par
```

Usar paréntesis (de ser necesario)

Agregar paréntesis extra para separar operaciones lógicas.

```
if (a == 2 and b != 3 or c == 1):
  ...
```

```
>> if ((a == 2) and ((b != 3) or (c == 1))):
  ...
```

Comentarios

Usar los comentarios para documentar:

- Líneas muy complejas (quizás convenga separarlas).
- Dejar la fuente de cierto código
- Documentar precondiciones y postcondiciones de funciones

No utilizarlos para:

- Código que no se usa
- Líneas que no necesitan explicación.

```
#asigno 3 a C
C = 3

# Esto no es mas necesario
# if (C == 2):
#  return C
```

```
# Recibe un numero entero y retorna un
booleano.
def es_par(n):
  ...

# El siguiente codigo fue extraido de ...
...
```

Simple frente a complicado

Simplificar el código nos permite:

- Hacerlo más legible
- Hacerlo más claro
- Hacerlo más extensible

No utilizar variables extra

No utilizar variables extra exceptuando aquellas que faciliten la lectura

```
def es_par(n):
  par = n % 2 == 0
  return par


def f(...):
  if (a == 0):
    return True
  return False
```

```
def es_par(n):
  return n % 2 == 0


def f(...):
  return a == 0
```

Evitar la programación spaguetti

Es importante para facilitar la lectura y hacer el código más escalable y reutilizable, no tener una función enorme que haga todo, si no separar en funciones que sean llamadas por esta función principal.

Las funciones separadas nos permiten reutilizarlas luego.

NO al código duplicado

El código duplicado es codigo igual que está presente en distintos lugares del código, por ejemplo al copiar y pegar codigo. Tener este tipo de código es algo MUY malo ya que si hay algún error, vamos a tener que corregirlo en todos los lugares donde copiamos. Un problema similar nos encontramos si necesitamos modificarlo.

La solución es simple: Extraer esas líneas a una función auxiliar y llamarla en todos lados donde se la necesita.

NO al código duplicado


```
def sumar_maximo(vector):
 maximo = vector[0]
 for i in vector:
   if (i > maximo):
     maximo = i
 return [i + maximo for i in vector]

# Resta el valor maximo a cada elemento
def sumar_maximo(vector):
 maximo = vector[0]
 for i in vector:
   if (i > maximo):
     maximo = i
 return [i - maximo for i in vector]
```

NO al código duplicado


```
# Suma el valor maximo a cada elemento
def buscar_maximo(vector):
 maximo = vector[0]
 for i in vector:
   if (i > maximo):
     maximo = i
 return maximo

def sumar_maximo(vector):
 maximo = buscar_maximo(vector)
 return [i + maximo for i in vector]

# Resta el valor maximo a cada elemento
def restar_maximo(vector):
 maximo = buscar_maximo(vector)
 return [i - maximo for i in vector]
```