NOMBRE:
Estándar de desarrollo de aplicaciones en Python

OBJETIVO:
Definir los lineamientos de codificación para el desarrollo de aplicaciones en Python

JUSTIFICACIÓN:

Este documento presenta la información sobre los lineamientos de codificación que son establecidos para el desarrollo de aplicaciones en Python

CATEGORÍA:
Desarrollo.

CUMPLIMIENTO REGULATORIO:
No aplica.

DESCRIPCIÓN DEL ESTÁNDAR:

A continuación, se detallan cada uno de los lineamientos definidos para el Grupo Bancolombia con el fin de implementarlos en el desarrollo de aplicaciones en Python

Convenciones Nombramiento

VARIABLES

- El nombre de las variables debe estar en inglés.
- El nombre de las variables no debe superar los 35 caracteres.
- Para el nombre de las variables la primera letra de cada palabra debe estar en mayúscula, exceptuando la primera letra de la variable (CamelCase - lower camel case).
- Las variables deben empezar por la letra “v”.
  Ejemplo:

```
vDataBaseName = “S_Productos”
vTableName = “S_Productos”
```

- Utilizar nombres claros y específicos, indicando puntualmente a que corresponde el nombramiento, primando que el nombre indicado caracterice la variable, para que no se genere ningún tipo de confusión a la hora de interpretar el código.
  Ejemplo:

```
a = ”19890518” -> MAL
vBirthday = ”19890518” -> BIEN

x = ”Jon Snow” -> MAL
vName = ”Jon Snow” -> BIEN
```

FUNCIONES

- El nombre de las funciones debe estar en inglés.
- El nombre de las funciones no debe superar los 35 caracteres.
- Para el nombre de las funciones, se debe separar por guion bajo (_) las palabras y debe estar todo en minúscula.
- Las variables que recibe el método no deben empezar por “v” pero si debe cumplir con el “CamelCase - lower camel case”.
- Las funciones deben empezar por el prefijo “fn” y
  Ejemplo:

```
fn_find_last_ingestion(tableName, dataBaseName)
```

```
fn_get_variable_get_context(workFlowName)
```

CLASES

- El nombre de las clases debe estar en inglés.
- El nombre de las clases no debe superar los 35 caracteres.
- Para el nombre de las clases la primera letra de cada palabra debe estar en mayúscula (CamelCase - upper camel case – Pascal Case)
  Ejemplo:

  ```
  class ModuleNameClassName:
  ```

  El "`ModuleName`" hace referencia al nombre del módulo sobre el cual se está implementando la clase.

  Ejemplo Práctico:

  ModuleName:

  ```
  beeline.py
  ```

  Class:

  ```
  class BeelineSubprocess:
  ```

Lineamientos

FUNCIONES

- Las funciones deben ser específicas y con una acción puntual, evitar desarrollar varias funcionalidades en una misma función, en caso tal de requerir una funcionalidad adicional, se debe crear una nueva función.

- La primera palabra en el nombre de las funciones debe ser un verbo en infinitivo, lo que nos permitirá identificar fácilmente que acción se está ejecutando en esa función.
  Ejemplo:

  - `fn_find_last_ingestion(tableName, dataBaseName)`

  - `fn_get_variable_get_context(workFlowName)`

- Se debe evitar el uso de funciones con muchos parámetros, procurar que el máximo de parámetros recibidos sea 5. En caso tal de requerir más de 5 argumentos se recomienda pasar una lista/vector o un archivo de propiedades/configuración (ConfigParser) como argumento a la función que se está implementando y dentro de la función hacer la separación de variables que contiene dicha lista o archivo.
  Ejemplo:

```
fn_find_last_ingestion(parametersList):
      vParametersListLength = len(parametersList)
      vArg1 = parametersList[0]
      vArg2 = parametersList[1]
      ...
      vArgN = parametersList[vParametersListLength - 1]

      do something

      return someThing

vListName = ['arg1', 'arg2', 'arg3', 'arg4', 'arg5', 'arg6']

vResultName = fn_find_last_ingestion(vListName)
```

- Usar un único return por función en lo posible. El uso del return por función colocado consecuentemente al final de esta facilita tanto la depuración como la adaptabilidad de los programas, este es uno de los principios de la programación estructurada. Dependiendo del flujo de las funciones puede darse que haya varios returns, se debe evitar funciones muy complejas, pero sentencias condicionales o de control se pueden usar para retornar

diferentes valores. Siempre se debe retornar un valor, en caso de tener una condición que no retorne nada, debe ser de la siguiente manera:
Ejemplo:

```
if vCondition1 == "Condition 1":
     return "Something1"
elif vCondition2 == "Condition 2":
     return "Something2"
elif vCondition3 == "Condition 3":
     return "Something3"
else:
     return -> MAL
-----------------------------------------
if vCondition1 == "Condition 1":
     return "Something1"
elif vCondition2 == "Condition 2":
     return "Something2"
elif vCondition3 == "Condition 3":
     return "Something3"
else:
     return None -> BIEN
```

VARIABLES

- Todas las variables deben ser inicializada con algún valor por defecto o el tipo de dato que contendrá esa variable.
  Ejemplo:

  - `vCustomerId = None`
  - `vCount = 0`
  - `vDictCncStrings = dict()`
  - `vListCustomersIds = list()`

GENERALES

- Para determinar prefijos y sufijos se deben utilizar las funciones startswith() y endswith(), esto nos permite tener mayor legibilidad del código.
  Ejemplo startswith():

```
vName = "Jon Snow"

if vname.startswith('Jon'):
     return True
else:
     return False
```

  Ejemplo endswith():

```
vName = "Jon Snow"

if vname.endswith('Snow'):
     return True
else:
     return False
```

- Mantener el mínimo número de niveles en las instrucciones anidadas, procurar que este no sea mayor a 3.
  Ejemplo:

```
if vCondition1 == "Condition 1":

     if vCondition2 == "Condition 2":

          if vCondition3 == "Condition 3":
               do something -> BIEN
continue coding
```

```
if vCondition1 == "Condition 1":

     if vCondition2 == "Condition 2":

          if vCondition3 == "Condition 3":

               if vCondition4 == "Condition 4":
                    do something -> MAL
```

```
continue coding
```

- Se debe desarrollar modularmente, es decir, creando funciones y evitar la duplicidad de código.

- Para evitar acceder a posiciones situadas fuera de los límites de un array se recomienda revisar si la expresión numérica usada para acceder a una posición determinada del array podría tomar algún valor que se esté fuera de los límites de este, si esto llegara a ocurrir podría generar una excepción en tiempo de ejecución entre otros acontecimientos, lo cual en lo posible debe ser controlado.
  Ejemplo:

```
aArrayName = ['Jon', 'Rob', 'Sansa']

try:
     print aArrayName[3]

except ValueError:

     print "That was not a valid index for the
     array aArrayName.  Try again..."
```

- No se debe tener datos quemados dentro del código que puedan ser susceptibles de algún cambio en el tiempo. El código debe estar en función de variables y en su lugar se deben llevar estos datos a archivos de configuración/propiedades externo a la aplicación que se está implementando, o bien pueden ser entregados en forma de parámetros (ConfigParser).
  Ejemplo:

```
vDataBaseCncStr = "jdbc:db2://serverName:5021/" -> MAL
```

```
import ConfigParser

vConfig = ConfigParser.ConfigParser()

vConfig.read("/path/file.config")

vStrCnxDataBase = vConfig.get('DataBase', 'CncStr') -> BIEN
```

- Ubicar todas las funciones y variables que apliquen en la parte superior del archivo.

- Para determinar si las cadenas están vacías o no se debe validar de la siguiente manera:

```
if not some_string:
```

```
if some_string:
```

- Todas las variables suelen y deben estar declaradas en el mismo lugar, la declaración de variables locales de un método debe hacerse en un bloque superior al método y deben ser inicializadas.
  Ejemplo:

```
fn_find_last_ingestion(arg1, arg2):
     vName = "Jon"
     vLastName = "Snow"

     do something ...

     vFullName = vName + " " + vLastName -> MAL

     do something ...

     return something
```

```
fn_find_last_ingestion(arg1, arg2):
     vName = "Jon"
     vLastName = "Snow"
     vFullName = vName + " " + vLastName -> BIEN

     do something ...

     return someThing
```

- Tener especial cuidado con el uso de los caracteres especiales en los comentarios, en caso de utilizar dichos caracteres se debe de tener en cuenta el encoding adecuado al inicio del código (# -*- coding: utf-8 -*-)

- El uso de Try-Catch es obligatorio si hay la probabilidad de que una excepción se genere y no pueda evitarse, de esta manera tenemos un control de expresiones en nuestro código y registramos en el LOG la causa clara del fallo.
  Ejemplo1:

```
aArrayName = ['Jon', 'Rob', 'Sansa']

try:
     print aArrayName[3]

except ValueError:

     print "That was not a valid index for the
     array aArrayName.  Try again..."
```

Ejemplo2:

```
vAge = "25"

try:
     vAgeNumber = int(vAge)

except ValueError:

     print "Error Casting to number"
```

Registro de eventos en el LOG

- Todos los programas deben dejar un LOG de su ejecución.
- Para el registro de logs se debe utilizar el módulo de Python “import logging”. Para más información en la implementación de este módulo por favor remitiré al siguiente link:

  https://docs.python.org/2/howto/logging.html

- El registro en el LOG debe cumplir con lo siguiente:

  INFORMACION:

  ```
  YYYY-MM-DD HH24:MM:SS - [INFO] – Mensaje claro de la acción ejecutada
  ```

  ADVERTENCIA:

  ```
  YYYY-MM-DD HH24:MM:SS - [WARNING] – Mensaje claro que indique puntualmente cual es la advertencia generada.
  ```

  ERROR:

  ```
  YYYY-MM-DD HH24:MM:SS - [ERROR] – Mensaje claro de la causa del error – Traza del error arrojada por el programa
  ```

- En el LOG no se debe registrar información sensible que pueda comprometer de alguna manera la seguridad de usuarios, clientes o del banco en sí mismo. Se debe evitar registrar nombres de usuario, contraseñas o cualquier otro tipo de información que pueda ser utilizada de manera maliciosa.

Comentarios

- Todos los scripts deben tener el siguiente encabezado:

```
"""
#--------------------------------------------------------------------------
# Author: Name LastName #CD: 02/02/2018 #LUD:
# Description: Description of the script
#
# Run: python scriptName.py arg1 arg2
#
# v0.1
# Modification:
# Description:
#--------------------------------------------------------------------------
"""
```

CD: Fecha de creación (Creation Date)
LUD: Fecha última modificación (Last Updated Date)
V0.1: Versión script

Los dos campos siguientes, hacen referencia a la modificación hecha en el script y la descripción de esta.

- Las funciones deben ser comentadas de la siguiente manera:

```
"""
#--------------------------------------------------------------------------
#----- Function to print in the log depending on the type of message -----
#---------------- @author: Name LastName (UserName) ----------------------
#
# inputs:
#       - logType: String with the message type (INFO, WARN, ERROR)
# output:
#       -
#
# usage:
#       print_log_messages(String)
#--------------------------------------------------------------------------
"""
def fn_print_log_messages(logType)
```

- Los comentarios solo deben ir sobre funciones, clases y/o paquetes. Se deben evitar los comentarios inline.
  Ejemplo:

  - `# Variable with the data base engine` -> MAL
    `vDataBaseEngineName = "Oracle";`

  - `# Variable with the user id`
    `vUserId = "56773334"`

  Este tipo de comentarios son excesivos.

  En algunas ocasiones se pueden estar utilizando librerías externas que no son de uso común o la complejidad del proceso implementado es muy alta. En estos casos es permitido comentarios inline.

En caso de utilizar librerías externas, se deben verificar con el área de infraestructura la viabilidad en cuanto a la instalación y uso de éstas en la LZ.

- Los comentarios deben estar a la par del código, es decir que han de utilizar el mismo nivel de indentación.
  Ejemplo1:

```
# Business logic to define monthly expenses per customer -> MAL
      if vCondition1 == "Condition 1": -> MAL

            if vCondition2 == "Condition 2":

                  if vCondition3 == "Condition 3":
                        do something

      continue coding ...
```

```
# Business logic to define monthly expenses per customer -> BIEN
if vCondition1 == "Condition 1": -> BIEN

      if vCondition2 == "Condition 2":

            if vCondition3 == "Condition 3":
                  do something ...

continue coding ...
```

Ejemplo2:

```
vCustomersSegments = [34455, 67564, 98724, 12987]

      # Business logic to identify VIP customers -> MAL
for customerSegment in vCustomersSegments:

      if customerSegment == someNumber
            do something ...
      else:
            do something else ...

      continue coding ...
```

```
vCustomersSegments = [34455, 67564, 98724, 12987]

# Business logic to identify VIP customers -> BIEN
for customerSegment in vCustomersSegments:

      if customerSegment == someNumber
            do something ...
      else:
            do something else ...

      continue coding ...
```

- Si se debe utilizar lógica de negocio, se debe documentar correctamente con suficientes comentarios.
  Ejemplo1:

```
# Business logic to define monthly expenses per customer
if vCondition1 == "Condition 1":

      # If Condition 2 do something
      if vCondition2 == "Condition 2":

            if vCondition3 == "Condition 3":

                  # Calculations to define monthly expenses
                  do something ...

            # If Condition3 is not applied, then do something else
            else:
                  do something else ...

continue coding ...
```

  Ejemplo2:

```
vCustomersSegments = [34455, 67564, 98724, 12987]

# Business logic to identify VIP customers
for customerSegment in vCustomersSegments:

      # If client segment equals to someNumber could be VIP
      if customerSegment == someNumber

            # Calculations to define if the customer is VIP
            do something ...
      else:
            do something else ...

      continue coding ...
```

- Corroborar ortografía, gramática y semántica de los comentarios, asegurándose que la puntuación es utilizada correctamente.

Indentación, Espaciado y agrupación de código

- Se recomienda tener una indentación de 4 espacios por nivel de anidamiento.

- Debe existir un espacio entre la declaración de la variable, el operador y su asignación.
  Ejemplo:

  - `vBirthday=”19890518”` -> MAL
    `vBirthday = ”19890518”` -> BIEN

  - `vCustomerFullName=”Jon Snow”` -> MAL
    `vCustomerFullName = ”Jon Snow”` -> BIEN

- Igualmente, para las funciones y concatenaciones, debe existir un espacio entre operaciones.
  Ejemplo:

  - `vSomething=“Hola”+“ Mundo”` -> MAL
    `vSomething = “Hola” + “ Mundo”` -> BIEN

  - `fn_something(variable1,variable2,variable3)` -> MAL
    `fn_something(variable1, variable2, variable3)` -> BIEN

- Es recomendable utilizar una o dos líneas en blanco para separar grupos lógicos de códigos.
  Ejemplo:

```
vCustomerName = “Jon”
vCustomerLastName = “Snow”
vCustomerId = 123456789
vCustomerAge = 25


fn_clean_linux_out_puts(linuxOutPut):
     some code ...
     some code ...

fn_find_last_ingestion(arg1, arg2, arg3):
     some code ...
     some code ...


continue coding ...
```

- Una buena práctica para separar una función de otra en una es utilizando una sola línea en blanco.

Ejemplo:

```
"""
#Function Comments
"""
fn_clean_linux_out_puts(linuxOutPut):
     some code ...
     some code ...

"""
#Function Comments
"""
fn_find_last_ingestion(arg1, arg2, arg3):
     some code ...
     some code ...

"""
#Function Comments
"""
fn_get_customer_monthly_expenses(arg1, arg2, arg3)
     some code ...
     some code ...
```

Módulos/Librerías Recomendadas (Buenas Prácticas)

CONFIGPARSER

- Se recomienda implementar esta librería en caso de requerir utilizar archivos de propiedades o de configuración. Remitirse a la siguiente documentación para entender su implementación:

  - Documentación Python2: https://docs.python.org/2/library/configparser.html

  - Documentación Python3.X: https://docs.python.org/3.5/library/configparser.html

  - Ejemplos Implementación: https://wiki.python.org/moin/ConfigParserExamples

DATETIME

- Se recomienda implementar esta librería en caso de requerir utilizar fechas y horas. Remitirse a la siguiente documentación para entender su implementación:

  - Documentación Python: https://docs.python.org/2/library/datetime.html#datetime.datetime

PARTES INTERESADAS:
Todas las personas que desempeñan un rol de desarrollo de soluciones en el Grupo Bancolombia, para los procesos COBIT:

Área	Rol
Dirección de Soporte y Calidad de Aplicaciones Dirección de Transformación. Proveedores de Desarrollo.	E: Responsable de ejecutar el estándar
Dirección de Soporte y Calidad de Aplicaciones Dirección de Transformación. Proveedores de Desarrollo. Dirección de Arquitectura	I: Requiere ser Informados
Dirección de Soporte y Calidad de Aplicaciones Dirección de Certificación	V: Responsable de verificar el estándar

EJECUCIÓN:

Subproceso: AI2 - Adquirir y mantener el software aplicativo.
Procedimiento: 01 – Atender Requerimientos Urgentes.
Procedimiento: 02 – Desarrollar e implementar soluciones.

VERIFICACIÓN

Subproceso: AI2 - Adquirir y mantener el software aplicativo.
Procedimiento: 01 – Atender Requerimientos Urgentes.
Procedimiento: 02 – Desarrollar e implementar soluciones.

La evidencia es el código fuente entregado por el proveedor.

EXCEPCIONES DE CONTROL:
No aplica.

GLOSARIO DE TÉRMINOS:

- **CamelCase**: es la práctica de escribir palabras o frases compuestas de tal manera que cada palabra o abreviatura en el medio de la frase comienza con una letra mayúscula, sin espacios intermedios ni signos de puntuación. Ejemplos comunes incluyen "iPhone", "eBay", "FedEx", "DreamWorks", "HarperCollins", entre otros.[1]

- **String**: En programación un String hace referencia a una variable de tipo texto.

- **startswith():** Hace referencia a una función propia de Python que nos permite validar si un String inicia con el valor que yo le paso a la función.

- **endswith ():** Hace referencia a una función propia de Python que nos permite validar si un String termina con el valor que yo le paso a la función.

- **Indentación**: Hace referencia a mover un bloque de texto hacia la derecha con espacios para diferenciarlo de texto paralelo.

- **Try Catch**: Es una instrucción de código usada comúnmente en los diferentes lenguajes de programación para el control de excepciones. El “Try” es la parte donde yo le indico que quiero hacer y el “Catch” es donde yo capturo la posible excepción que desencadeno la acción que intente ejecutar en el “Try”.

- **LOG:** Archivo donde se deja registro/evidencia de la ejecución del programa que permita analizar en caso de falla la posible causa de esta.

[1] https://en.wikipedia.org/wiki/Camel_case

REVISIONES:

- **Ciclo de revisión:** Anual
- **Fecha de creación:** 2018/02/01
- **Fecha inicio de vigencia:** 2018/02/01
- **Fecha de última revisión:** 2018/02/01

HISTORIAL DE VERSIONES

Versión	Área	Participante	Observaciones/Descripción
1.0	SECCION SOPORTE INFORMACION	Juan Esteban Castaño Salazar	

HISTORIAL DE APROBACIÓN

Versión	Aprueban	Responsable	Fecha de Aprobación
1.0	Comité de Expertos	David Esteban Echeverri	2017/02/08