Adrianna Alvarez & Ava DeCristofaro
G87.2026.T09.GE1

1. # Naming style matching correct argument names: Argument names must follow camelCase naming convention.

CORRECT USAGE:

```
def ValidateCIF( self, cIf ):
```

INCORRECT USAGE:

```
def ValidateCIF( self, CiF ):
```

PYLINT:

```
# Naming style matching correct argument names.
#MODIFIED RULE
#argument-naming-style=snake_case
argument-naming-style=camelCase
```

2. # Naming style matching correct attribute names: Attribute names must follow camelCase naming convention.

CORRECT USAGE:

```
@property
    def enterpriseCIf(self):
        return self.cIf
```

INCORRECT USAGE:

```
@property
    def enterprise-cif(self):
        return self.cIf
```

PYLINT:

```
# Naming style matching correct attribute names.
#MODIFIED RULE
#attr-naming-style=snake_case
attr-naming-style=camelCase
```

3. # Naming style matching correct variable names: Variable names must follow the UPPER_CASE naming convention.

CORRECT USAGE:

```
EVEN_SUM = int(DIGITS[1]) + int(DIGITS[3]) + int(DIGITS[5])
```

INCORRECT USAGE:

```
evenSum = int(digits[1]) + int(digits[3]) + int(digits[5])
```

PYLINT:

```
# Naming style matching correct variable names.
#MODIFIED RULE
#variable-naming-style=snake_case
variable-naming-style=UPPER_CASE
```

4. # Naming style matching correct constant names: Constants must follow the snake_case naming convention.

CORRECT USAGE:

```
em = EnterpriseManager()
```

INCORRECT USAGE:

```
EM = EnterpriseManager()
```

PYLINT:

```
# Naming style matching correct constant names.
#MODIFIED RULE
#const-naming-style=UPPER_CASE
const-naming-style=snake_case
```

5. # Naming style matching correct module names: Module names must follow the PascalCase naming convention

CORRECT USAGE:

```
EnterpriseManager.py
```

INCORRECT USAGE:

```
enterpriseManager.py
```

PYLINT:

```
# Naming style matching correct module names.
#MODIFIED RULE
#module-naming-style=snake_case
module-naming-style=PascalCase
```

6. # Naming style matching correct method names: Method names must follow the PascalCase naming convention.

CORRECT USAGE:

```
def ValidateCIF( self, CiF ):
```

INCORRECT USAGE:

```
def validate-cif( self, CiF ):
```

PYLINT:

```
# Naming style matching correct method names.
#MODIFIED RULE
#method-naming-style=snake_case
method-naming-style=PascalCase
```

7. # Minimum line length for functions/classes that require docstrings, shorter ones are exempt: Functions and classes that are > 10 characters long must have a docstring.

CORRECT USAGE:

```
class EnterpriseManager:
    """
    Manages enterprise-related operations such as CIF validation
    and request creation from JSON input.
    """
```

INCORRECT USAGE:

```
class EnterpriseManager:
```

PYLINT:

```
# Minimum line length for functions/classes that require
docstrings, shorter
# ones are exempt.
#MODIFIED RULE
#docstring-min-length=-1
docstring-min-length=10
```

8. # Maximum number of locals for function / method body: A function/method cannot have more than 16 local variables within its body.

CORRECT USAGE:

```
def ValidateCIF( self, CiF ):
```

```
    """
    Validate a CIF identifier.

    Returns True if the CIF is valid according to format and
    checksum rules, otherwise False.
    """
    # PLEASE INCLUDE HERE THE CODE FOR VALIDATING THE GUID
    # 1) Basic format: [Letter][7 digits][Control]
    if not isinstance(CiF, str):
        return False

    CIF = CiF.strip().upper()
    if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
        return False

    LETTER = CIF[0]
    DIGITS = CIF[1:8]  # 7-digit block
    CONTROL = CIF[8]  # last char

    # 2) Step 1: sum digits in even positions of the block
(positions 2,4,6)
    # positions are 1..7 left-to-right, so indices 1,3,5
    EVEN_SUM = int(DIGITS[1]) + int(DIGITS[3]) + int(DIGITS[5])

    # 3) Step 2: odd positions (1,3,5,7) -> multiply by 2, sum
digits if needed
    ODD_SUM = 0
    for IDX in (0, 2, 4, 6):
        V = int(DIGITS[IDX]) * 2
        ODD_SUM += (V // 10) + (V % 10)  # digit-sum (0..18)

    # 4) Step 3: partial sum
    PARTIAL_SUM = EVEN_SUM + ODD_SUM

    # 5) Step 4: base digit
    UNITS = PARTIAL_SUM % 10
    BASE_DIGIT = (10 - UNITS) % 10  # handles "if units is 0 ->
base digit is 0"

    # 6) Step 5: control character rule + mapping table
    BASE_TO_LETTER = {0: "J", 1: "A", 2: "B", 3: "C", 4: "D",
                      5: "E", 6: "F", 7: "G", 8: "H", 9: "I"}
```

```
    EXPECTED_DIGIT = str(BASE_DIGIT)
    EXPECTED_LETTER = BASE_TO_LETTER[BASE_DIGIT]

    # Letter-based rules you provided
    if LETTER in ("A", "B", "E", "H"):
        return CONTROL == EXPECTED_DIGIT

    if LETTER in ("K", "P", "Q", "S"):
        return CONTROL == EXPECTED_LETTER

    # For other CIF letters, accept either (common CIF behavior)
    return (CONTROL == EXPECTED_DIGIT) or (CONTROL ==
EXPECTED_LETTER)
    # RETURN TRUE IF THE GUID IS RIGHT, OR FALSE IN OTHER CASE
```

INCORRECT USAGE:

```
def ValidateCIF( self, CiF ):
    """
    Validate a CIF identifier.

    Returns True if the CIF is valid according to format and
    checksum rules, otherwise False.
    """
    # PLEASE INCLUDE HERE THE CODE FOR VALIDATING THE GUID
    # 1) Basic format: [Letter][7 digits][Control]
    if not isinstance(CiF, str):
        return False

    CIF = CiF.strip().upper()
    if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
        return False

    EXTRA_LOCAL_VAR = CIF[0] # num of local vars > 16
    LETTER = CIF[0]
    DIGITS = CIF[1:8]  # 7-digit block
    CONTROL = CIF[8]  # last char

    # 2) Step 1: sum digits in even positions of the block
(positions 2,4,6)
    # positions are 1..7 left-to-right, so indices 1,3,5
```

```
    EVEN_SUM = int(DIGITS[1]) + int(DIGITS[3]) + int(DIGITS[5])

    # 3) Step 2: odd positions (1,3,5,7) -> multiply by 2, sum
digits if needed
    ODD_SUM = 0
    for IDX in (0, 2, 4, 6):
        V = int(DIGITS[IDX]) * 2
        ODD_SUM += (V // 10) + (V % 10)  # digit-sum (0..18)

    # 4) Step 3: partial sum
    PARTIAL_SUM = EVEN_SUM + ODD_SUM

    # 5) Step 4: base digit
    UNITS = PARTIAL_SUM % 10
    BASE_DIGIT = (10 - UNITS) % 10  # handles "if units is 0 ->
base digit is 0"

    # 6) Step 5: control character rule + mapping table
    BASE_TO_LETTER = {0: "J", 1: "A", 2: "B", 3: "C", 4: "D",
                      5: "E", 6: "F", 7: "G", 8: "H", 9: "I"}

    EXPECTED_DIGIT = str(BASE_DIGIT)
    EXPECTED_LETTER = BASE_TO_LETTER[BASE_DIGIT]

    # Letter-based rules you provided
    if LETTER in ("A", "B", "E", "H"):
        return CONTROL == EXPECTED_DIGIT

    if LETTER in ("K", "P", "Q", "S"):
        return CONTROL == EXPECTED_LETTER

    # For other CIF letters, accept either (common CIF behavior)
    return (CONTROL == EXPECTED_DIGIT) or (CONTROL ==
EXPECTED_LETTER)
    # RETURN TRUE IF THE GUID IS RIGHT, OR FALSE IN OTHER CASE
```

PYLINT:

```
# Maximum number of locals for function / method body.
#MODIFIED RULE
#max-locals=15
max-locals=16
```

9. # Maximum number of public methods for a class (see R0904): A class cannot have more than 7 public methods

CORRECT USAGE:

```
class OrderProcessor:
    def __init__(self):
        pass

    def create(self):
        pass

    def update(self):
        pass

    def delete(self):
        pass

    def validate(self):
        pass

    def save(self):
        pass
```

INCORRECT USAGE:

```
class OrderProcessor:
    def __init__(self):
        pass

    def create(self):
        pass

    def update(self):
        pass

    def delete(self):
        pass

    def validate(self):
        pass
```

```
    def save(self):
        pass

    def load(self):
        pass

    def export(self):
        pass
```

PYLINT

```
# Maximum number of public methods for a class (see
R0904).
#MODIFIED RULE
#max-public-methods=10
max-public-methods=7
```

10. # Maximum number of characters on a single line. Pylint's default of 100 is based on PEP 8's guidance that teams may choose line lengths up to 99 characters.

CORRECT USAGE:

```
total_price = (
    base_price
    + tax_amount
    + shipping_cost
    + handling_fee
    + insurance_fee
)
```

INCORRECT USAGE:

```
total_price = base_price + tax_amount + shipping_cost +
handling_fee + insurance_fee
```

PYLINT:

```
# Maximum number of characters on a single line.
Pylint's default of 100 is
# based on PEP 8's guidance that teams may choose line
lengths up to 99
# characters.
#MODIFIED RULE
```

```
#max-line-length=100
max-line-length=99
```