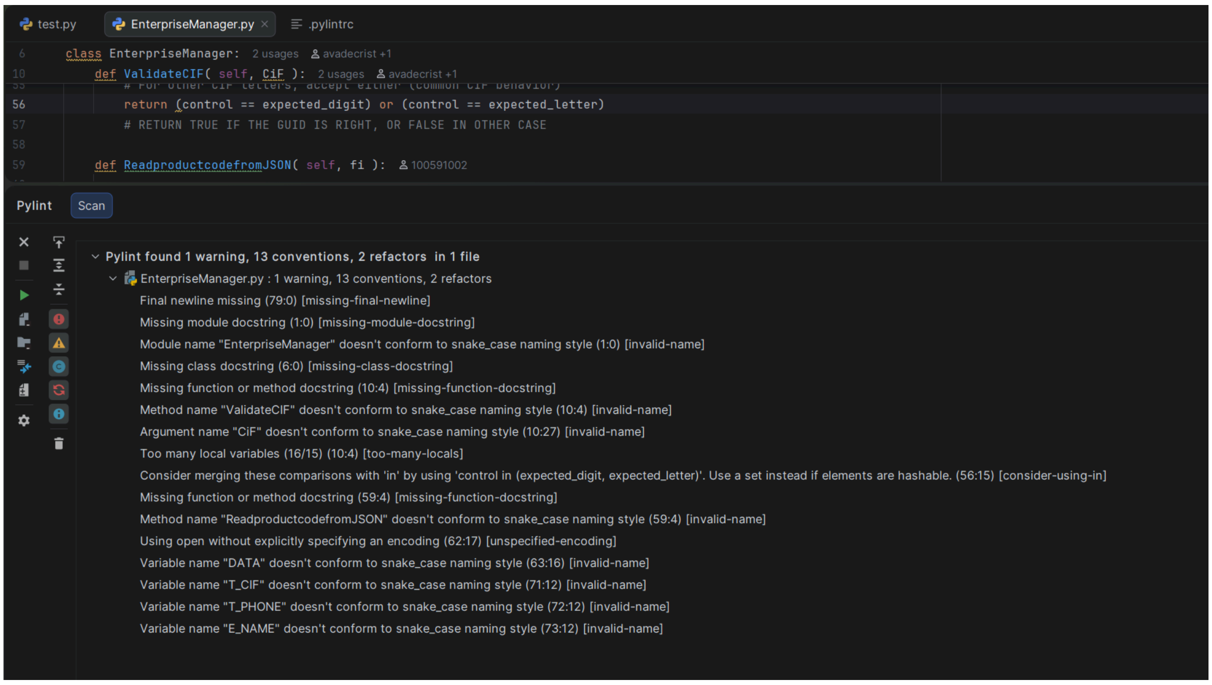
Adrianna Alvarez & Ava DeCristofaro
G87.2026.T09.GE1

BEFORE:



```
class EnterpriseManager:  2 usages  avadecrist +1
    def ValidateCIF( self, CiF ):  2 usages  avadecrist +1
        return (control == expected_digit) or (control == expected_letter)
        # RETURN TRUE IF THE GUID IS RIGHT, OR FALSE IN OTHER CASE

    def ReadproductcodefromJSON( self, fi ):  100591002
```

```
Pylint found 1 warning, 13 conventions, 2 refactors  in 1 file
  EnterpriseManager.py : 1 warning, 13 conventions, 2 refactors
    Final newline missing (79:0) [missing-final-newline]
    Missing module docstring (1:0) [missing-module-docstring]
    Module name "EnterpriseManager" doesn't conform to snake_case naming style (1:0) [invalid-name]
    Missing class docstring (6:0) [missing-class-docstring]
    Missing function or method docstring (10:4) [missing-function-docstring]
    Method name "ValidateCIF" doesn't conform to snake_case naming style (10:4) [invalid-name]
    Argument name "CiF" doesn't conform to snake_case naming style (10:27) [invalid-name]
    Too many local variables (16/15) (10:4) [too-many-locals]
    Consider merging these comparisons with 'in' by using 'control in (expected_digit, expected_letter)'. Use a set instead if elements are hashable. (56:15) [consider-using-in]
    Missing function or method docstring (59:4) [missing-function-docstring]
    Method name "ReadproductcodefromJSON" doesn't conform to snake_case naming style (59:4) [invalid-name]
    Using open without explicitly specifying an encoding (62:17) [unspecified-encoding]
    Variable name "DATA" doesn't conform to snake_case naming style (63:16) [invalid-name]
    Variable name "T_CIF" doesn't conform to snake_case naming style (71:12) [invalid-name]
    Variable name "T_PHONE" doesn't conform to snake_case naming style (72:12) [invalid-name]
    Variable name "E_NAME" doesn't conform to snake_case naming style (73:12) [invalid-name]
```

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:

```
# 1 (self), 2 (ciF)
def ValidateCIF( self, CiF ):
   if not isinstance(CiF, str):
       return False
# 3
   CIF = CiF.strip().upper()
   if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
       return False
# 4
   LETTER = CIF[0]
# 5
   DIGITS = CIF[1:8]
```

```
# 6
   CONTROL = CIF[8]

# 7
   EVEN_SUM = int(DIGITS[1]) + int(DIGITS[3]) +
int(DIGITS[5])

# 8
   ODD_SUM = 0
# 9 (variable IDX within loop)
   for IDX in (0, 2, 4, 6):
# 10 (variable V)
       V = int(DIGITS[IDX]) * 2
       ODD_SUM += (V // 10) + (V % 10)

# 11
   PARTIAL_SUM = EVEN_SUM + ODD_SUM
# 12
   UNITS = PARTIAL_SUM % 10
# 13
   BASE_DIGIT = (10 - UNITS) % 10
# 14
   BASE_TO_LETTER = {0: "J", 1: "A", 2: "B", 3: "C", 4:
"D",
                     5: "E", 6: "F", 7: "G", 8: "H", 9:
"I"}
# 15
   EXPECTED_DIGIT = str(BASE_DIGIT)
# 16
   EXPECTED_LETTER = BASE_TO_LETTER[BASE_DIGIT]

   if LETTER in ("A", "B", "E", "H"):
       return CONTROL == EXPECTED_DIGIT

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

   return (CONTROL == EXPECTED_DIGIT) or (CONTROL ==
EXPECTED_LETTER)

# 16 local vars <= 16 ✅
```

INCORRECT USAGE:

```
# 1 (self), 2 (ciF)
def ValidateCIF( self, CiF ):
   if not isinstance(CiF, str):
       return False
# 3
   CIF = CiF.strip().upper()
   if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
       return False
# 4
   LETTER = CIF[0]
# 5
   DIGITS = CIF[1:8]
# 6
   CONTROL = CIF[8]

# 7
   EVEN_SUM = int(DIGITS[1]) + int(DIGITS[3]) +
int(DIGITS[5])

# 8
   ODD_SUM = 0
# 9 (variable IDX within loop)
   for IDX in (0, 2, 4, 6):
# 10 (variable V)
       V = int(DIGITS[IDX]) * 2
       ODD_SUM += (V // 10) + (V % 10)

# 11
   PARTIAL_SUM = EVEN_SUM + ODD_SUM
# 12
   UNITS = PARTIAL_SUM % 10
# 13
   BASE_DIGIT = (10 - UNITS) % 10
# 14
   BASE_TO_LETTER = {0: "J", 1: "A", 2: "B", 3: "C", 4:
"D",
                     5: "E", 6: "F", 7: "G", 8: "H", 9:
"I"}
# 15
   EXPECTED_DIGIT = str(BASE_DIGIT)
# 16
   EXPECTED_LETTER = BASE_TO_LETTER[BASE_DIGIT]
   if LETTER in ("A", "B", "E", "H"):
       return CONTROL == EXPECTED_DIGIT

   if LETTER in ("K", "P", "Q", "S"):
       return CONTROL == EXPECTED_LETTER
# 17
   IS_VALID = CONTROL in (EXPECTED_DIGIT,
EXPECTED_LETTER)

   return IS_VALID

# 17 local vars > 16 ❌
```

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 3 public methods

CORRECT USAGE:

```
class EnterpriseManager:
# 1
   def ValidateCIF(self, cIf):
       # . . .

# 2
   def ReadproductcodefromJSON( self, fi ):
       # . . .
# 2 public methods < 3 ✅
```

INCORRECT USAGE:

```
class EnterpriseRequest:

   def __init__(self, cIf, phOnE, eName):
       self.enterpriseName = eName
```

```
        self.cIf = cIf
        self.phone = phOnE
        JUST_NOW = datetime.utcnow()
        self._timeStamp = datetime.timestamp(JUST_NOW)

    def __str__(self):
        return "Enterprise:" + json.dumps(self.__dict__)
# 1
    @property
    def enterpriseCIf(self):
        return self.cIf
    @enterpriseCIf.setter
    def enterpriseCIf(self, value):
        self.cIf = value
# 2
    @property
    def phoneNumber(self):
        return self.phone
    @phoneNumber.setter
    def phoneNumber(self, value):
        self.phone = value
# 3
    @property
    def enterpriseName(self):
        return self.enterpriseName
    @enterpriseName.setter
    def enterpriseName(self, value):
        self.enterpriseName = value

# 4 (extra public method)
    def validate(self):
        pass

# 4 public methods > 3   ❌
```

PYLINT:

```
# Maximum number of public methods for a class (see
R0904).
#MODIFIED RULE
#max-public-methods=10
max-public-methods=3
```

10. # Maximum number of characters on a single line: Limiting lines to 79 characters based on python's standard library.

CORRECT USAGE:

```
except FileNotFoundError as E:
    raise EnterpriseManagementException(
        "Wrong file or file path"
    ) from E
```

INCORRECT USAGE:

```
except FileNotFoundError as E:
   raise EnterpriseManagementException("Wrong file or
file path") from E
```

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=79
```

11. # Minimum number of public methods for a class (see R0903): There must be at least one public method per class

CORRECT USAGE:

```
class EnterpriseManagementException(Exception):
    """
    Exception raised for errors related to enterprise
management.

    Attributes:
    message (str): Human-readable error message
describing the failure.
    """
    def __init__(self, message):
        self.__message = message
        super().__init__(self.message)
```

INCORRECT USAGE:

```
class EnterpriseManagementException(Exception):
    """
    Exception raised for errors related to enterprise
management.

    Attributes:
    message (str): Human-readable error message
describing the failure.
    """
```

PYLINT:

```
# Minimum number of public methods for a class (see
R0903).
#MODIFIED RULE
#min-public-methods=2
min-public-methods=1
```

12. <u># This flag controls whether inconsistent-quotes generates a warning when the character used as a quote delimiter is used inconsistently within a module:</u> Ensures that the quotes are all either “ or ‘

CORRECT USAGE:

```
for e in EXAMPLES:
    print(e, "->", em.ValidateCIF(e))
print("all done!")
```

INCORRECT USAGE:

```
for e in EXAMPLES:
    print(e, "->", em.ValidateCIF(e))
print('all done!')
```

PYLINT:

```
# This flag controls whether inconsistent-quotes
generates a warning when the
# character used as a quote delimiter is used
inconsistently within a module.
```

```
#MODIFIED RULE
#check-quote-consistency=no
check-quote-consistency=yes
```

13. # Maximum number of arguments for function / method: A function cannot have more than 4 arguments.

CORRECT USAGE:

```
def __init__(self, cIf, phone, eName):
        self.enterpriseName = eName
        self.cIf = cIf
        self.phone = phone
        JUST_NOW = datetime.utcnow()
        self._timeStamp = datetime.timestamp(JUST_NOW)
```

INCORRECT USAGE:

```
   def __init__(self, cIf, phone, eName, extraArg):
        self.enterpriseName = eName
        self.cIf = cIf
        self.phone = phone
        JUST_NOW = datetime.utcnow()
        self._timeStamp = datetime.timestamp(JUST_NOW)
```

PYLINT:

```
# Maximum number of arguments for function / method.
#MODIFIED RULE
#max-args=5
max-args=4
```

14. # Maximum number of return / yield for function / method body: A method body cannot contain over 5 return statements.

CORRECT USAGE:

```
  def ValidateCIF(self, cIf):
        if not isinstance(cIf, str):
            return False #1
        CIF = cIf.strip().upper()
        if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
            return False #2

        LETTER = CIF[0]
        DIGITS = CIF[1:8]
        CONTROL = CIF[8]

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

        ODD_SUM = 0
        for IDX in (0, 2, 4, 6):
            V = int(DIGITS[IDX]) * 2
            ODD_SUM += (V // 10) + (V % 10)

        PARTIAL_SUM = EVEN_SUM + ODD_SUM

        UNITS = PARTIAL_SUM % 10
        BASE_DIGIT = (10 - UNITS) % 10

        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]

        if LETTER in ("A", "B", "E", "H"):
            return CONTROL == EXPECTED_DIGIT #3

        if LETTER in ("K", "P", "Q", "S"):
            return CONTROL == EXPECTED_LETTER #4

        return CONTROL in (EXPECTED_DIGIT,
EXPECTED_LETTER) #5
```

INCORRECT USAGE:

```
def ValidateCIF(self, cIf):
        if not isinstance(cIf, str):
            return False #1

        CIF = cIf.strip().upper()
        if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
            return False #2

        LETTER = CIF[0]
        DIGITS = CIF[1:8]
        CONTROL = CIF[8]

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

        ODD_SUM = 0
        for IDX in (0, 2, 4, 6):
            V = int(DIGITS[IDX]) * 2
            ODD_SUM += (V // 10) + (V % 10)

        PARTIAL_SUM = EVEN_SUM + ODD_SUM

        UNITS = PARTIAL_SUM % 10
        BASE_DIGIT = (10 - UNITS) % 10
        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]

        if LETTER in ("A", "B", "E", "H"):
            return CONTROL == EXPECTED_DIGIT #3

         if LETTER in ("K", "P", "Q", "S"):
            return CONTROL == EXPECTED_LETTER #4
        else:
            return CONTROL #5


        return CONTROL in (EXPECTED_DIGIT,
EXPECTED_LETTER) #6
```

PYLINT:

```
#Maximum number of return / yield for function / method
body.
#max-returns=6
max-returns=5
```

15. # Maximum number of branches for function / method body: A method body cannot contain more than 5 branches.

CORRECT USAGE:

```
def ValidateCIF(self, cIf):
    # 1
        if not isinstance(cIf, str):
            return False

        CIF = cIf.strip().upper()
    # 2
        if not re.fullmatch(r"[A-Z]\d{7}[A-Z0-9]", CIF):
            return False

        LETTER = CIF[0]
        DIGITS = CIF[1:8]
        CONTROL = CIF[8]


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


        ODD_SUM = 0
    # 3
        for IDX in (0, 2, 4, 6):
            V = int(DIGITS[IDX]) * 2
            ODD_SUM += (V // 10) + (V % 10)

        PARTIAL_SUM = EVEN_SUM + ODD_SUM


        UNITS = PARTIAL_SUM % 10
        BASE_DIGIT = (10 - UNITS) % 10
        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]

    # 4
        if LETTER in ("A", "B", "E", "H"):
            return CONTROL == EXPECTED_DIGIT
    # 5
        if LETTER in ("K", "P", "Q", "S"):
            return CONTROL == EXPECTED_LETTER

        return CONTROL in (EXPECTED_DIGIT,
EXPECTED_LETTER)
    # 5 branches <= 5 ✅
```

INCORRECT USAGE:

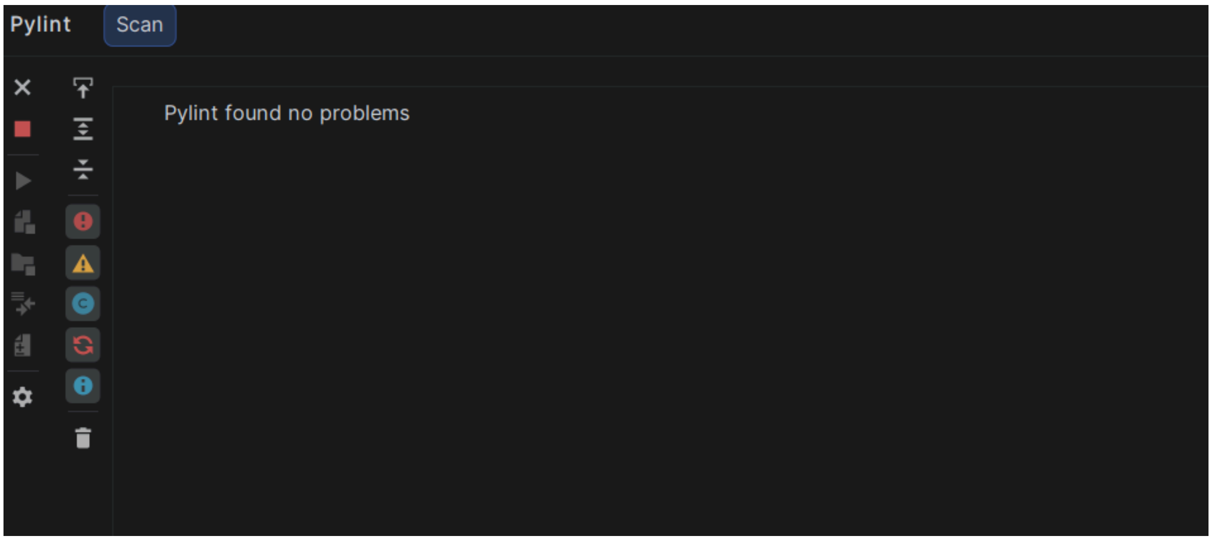
```
def ValidateCIF(self, cIf):
    # 1
        if not isinstance(cIf, str):
            return False

        CIF = cIf.strip().upper()
    # 2
        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


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


        ODD_SUM = 0
    # 3
```

```
        for IDX in (0, 2, 4, 6):
            V = int(DIGITS[IDX]) * 2
            ODD_SUM += (V // 10) + (V % 10)

        PARTIAL_SUM = EVEN_SUM + ODD_SUM


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


        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]

    # 4
        if LETTER in ("A", "B", "E", "H"):
            return CONTROL == EXPECTED_DIGIT
    # 5
        if LETTER in ("K", "P", "Q", "S"):
            return CONTROL == EXPECTED_LETTER
    # 6
        if not LETTER in ("A", "B", "E", "H", "K", "P",
"Q", "S"):
            print("invalid control")

        return CONTROL in (EXPECTED_DIGIT,
EXPECTED_LETTER)
    # 6 branches > 5 ❌
```

PYLINT:

```
# Maximum number of branch for function / method body.
#MODIFIED RULE
#max-branches=12
max-branches=5
```

AFTER:



Pylint found no problems