Commenting and Documentation

Comments

Readability and Pythonic code

Readability

Readability refers to how easily code can be read, understood, and maintained by humans. Python emphasizes readability more than many other languages.

1 .Meaningful Names:

- Variables, functions, and class names should describe their purpose.
  Avoid single letters except in loops (like `i` or `j`).

```
# Bad readability

x = 10

y = 20

z = x + y

# Good readability

width = 10

height = 20

area = width + height
```

2. Consistent Indentation and Spacing:

- Python uses indentation to define blocks; always be consistent.
- Follow **PEP 8** style guide: 4 spaces per indentation.

```
# Bad indentation

if x>10:

print("x is greater than 10")

# Good indentation

if x > 10:
    print("x is greater than 10")
```

3. Commenting and Documentation:

   - Use comments to explain *why* code is written, not *what* it does.

   - Use docstrings for functions/classes.

```
def calculate_area(radius):

    """

    Calculate the area of a circle given the radius.

    Formula: area = π * r^2

    """

    import math

    return math.pi * radius ** 2
```

4. Avoid Deep Nesting:

   - Too many nested loops or conditionals reduce readability.

   - Consider breaking code into functions.

5. Readable Structure:

   - Use blank lines to separate sections.

   - Keep code blocks small and organized.

Pythonic Code

Pythonic code is code that follows Python's conventions and idioms. It's not just about functionality; it's about writing code in a way that's natural for Python and takes advantage of its features.

Characteristics of Pythonic Code:

1. Use Built-in Functions:
   - Python provides many built-in functions that simplify tasks.

```
# Non-Pythonic
```

```
squares = []
for i in range(10):
    squares.append(i*i)
# Pythonic
squares = [i*i for i in range(10)]  # List comprehension
```

2. Prefer Readable Expressions:

- Use concise, clear syntax rather than verbose code.

```
# Non-Pythonic
if len(my_list) != 0:
    print("List has items")
# Pythonic
if my_list:
    print("List has items")
```

3. Follow “Easier to ask for forgiveness than permission” (EAFP):

- Instead of checking for conditions, try the operation and handle exceptions.

```
# Non-Pythonic
if 'key' in my_dict:
    value = my_dict['key']
# Pythonic
try:
    value = my_dict['key']
except KeyError:
    value = None
```

4. Use Generators and Iterators Where Appropriate:

- Saves memory and makes code elegant.

```
# Pythonic: generator expression

total = sum(x*x for x in range(10))
```

Functions and DRY Principles

1. Functions in Python

A function is a reusable block of code that performs a specific task.
Instead of writing the same logic multiple times, we can define a function once and call it whenever needed.

Key Parts of a Function:

1. Definition – created using `def` keyword.
2. Parameters – input values passed to the function.
3. Return Value – output produced by the function.
4. Calling – using the function when needed.

Example:

```
# Function definition

def greet(name):

    return f"Hello, {name}!"

# Function call

print(greet("Lekshmi"))

print(greet("Rahul"))
```

Output:

```
Hello, Lekshmi!

Hello, Rahul!
```

2. DRY Principle

DRY = *Don’t Repeat Yourself*

It is a **software design principle** that says:
"Every piece of knowledge must have a single, unambiguous, and authoritative representation in the system."

In simple words: **Don’t write the same code in multiple places. Write it once, and reuse it.**

Bad Example (Without DRY):

```
# Calculating area of rectangle in multiple places

length1, width1 = 5, 10

area1 = length1 * width1

print("Area 1:", area1)

length2, width2 = 7, 3

area2 = length2 * width2

print("Area 2:", area2)
```

Problem: The formula is repeated. If we need to change logic, we must edit it everywhere.

Good Example (With DRY using Functions):

```
def rectangle_area(length, width):

    return length * width

print("Area 1:", rectangle_area(5, 10))

print("Area 2:", rectangle_area(7, 3))
```

Advantages of the above code :

- Logic is written **once** inside the function.
- Easy to **reuse** and **modify** later.

Functions = Reusable code blocks → improve readability, modularity, and maintainability.

DRY Principle = *Don't Repeat Yourself* → avoid duplication, write code once, reuse many times.

Exception Handling and **Validation**

An **exception** is an error that occurs during program execution.

If not handled, it stops the program.

Eg: print(10 / 0) # ZeroDivisionError

This will crash the program

Exception Handling with `try-except`

Python provides `try-except` blocks to **catch errors** and prevent program crashes.

Eg:-

```
try:
   num = int(input("Enter a number: "))
   result = 10 / num
   print("Result:", result)
except ZeroDivisionError:
   print("Error: Cannot divide by zero!")
except ValueError:
   print("Error: Please enter a valid number.")
```

- If user enters `0` → Caught by `ZeroDivisionError`.
- If user enters `"abc"` → Caught by `ValueError`.

Adding `else` and `finally`

`else` → runs if no exception occurs.

`finally` → always runs (used for cleanup).

```
try:
    num = int(input("Enter a number: "))
    print("Square:", num * num)
except ValueError:
    print("Invalid input!")
else:
    print("No error occurred.")
finally:
    print("Program finished.")
```

Raising Exceptions

You can **manually raise exceptions** with `raise`.

```
def withdraw(amount):
    if amount < 0:
        raise ValueError("Amount cannot be negative")
    return f"Withdrew {amount} successfully"
print(withdraw(100))
print(withdraw(-50))  # Raises ValueError
```

Validation in Python

Validation means checking whether input or data is valid before using it.It prevents runtime errors and ensures correct results.

Example: Input Validation

```
age = input("Enter your age: ")
if age.isdigit() and int(age) > 0:
    print("Valid age:", age)

else:
    print("Invalid age entered!")
```

Example: Validation with Functions

```
def validate_email(email):
    if "@" in email and "." in email:
        return True
    return False
print(validate_email("test@example.com"))  # True
print(validate_email("invalid_email"))     # False
```

Example: Validation + Exception Handling

Both can work together:

```
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Both inputs must be numbers")
    if b == 0:
        raise ValueError("Denominator cannot be zero")
    return a / b
try:
    print(divide(10, 2))
    print(divide(10, 0))
except Exception as e:
    print("Error:", e)
```

Exception Handling

- Use try-except to catch errors.
- Use else for successful execution, finally for cleanup.
- Use raise to throw custom exceptions.

Validation

- Ensures input/data is correct before processing.
- Prevents errors early.
- Often used with exception handling.