Pycimen Language Reference

Syntax:

- Blocks are defined by indentation.
- Variable assignments use the `=` symbol.
- Expressions are generally the same as in Python.

Data Types:

Pycimen supports the basic data types:

- `int` - Integers
- `float` - Floating-point numbers
- `string` - String literals
- `boolean` - True and False
- `None` - Equivalent to Python's `None`

Operators:

Pycimen supports the following operators:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Comparison operators: `<`, `>`, `==`, `!=`, `<=`, `>=`
- Logical operators: `and`, `or`, `not`
- Bitwise operators: `&`, `|`, `^`, `<<`, `>>`

Control Flow:

Pycimen supports the following control flow statements:

- `if/elif/else`
- `while` loop
- `for` loop
- `break`
- `continue`
- `pass`

Functions:

Functions are defined with the `def` keyword and parameters are specified in parentheses.

Classes:

Pycimen supports class definition with the `class` keyword.

Other Features:

- The `print` statement works the same as in Python.
- The `return` statement is also used as in Python.

Notes:

- Pycimen does not currently support dictionaries, sets, and tuples as in Python.

1. Syntax Rules

1.1. Indentation

In Pycimen, code blocks are defined by indentation. Indentation can be created using spaces or tab characters, but mixed use within the same block is not allowed.

Correct Usage:

```
if x > 0:
   print("Positive")  # Correct
   print("Value")
```

Incorrect Usage (Mixed Indentation):

```
if x > 0:
   print("Positive")  # Incorrect! Mixed indentation
      print("Value")
```

1.2. Line Breaks

In Pycimen, many statements can be written on a single line, but for longer statements, multiple lines can be used. The backslash "" character is used for this purpose.

Example:

```
x = 1 + 2 + \
   3 + 4
```

1.3. Comment Lines

Single-line comments start with the # character.

Examples:

```
# Bu bir yorumdur
print("Merhaba") # Bu da bir yorum
```

1.4. Multiline Comments/Docstrings

Multiline comments or docstrings are enclosed in triple quotes (""" or ```).

Example:

```
"""
Bu bir
çok satırlı
docstringdir.
"""
```

2. Data Types

Pycimen supports the following basic data types:

Data Type	Description	Example
int	Represents whole numbers.	42, -100, 0
float	Represents numbers with decimal places.	3.14, -5.23, 1.7e10
str	Represents text enclosed in single or double quotes. Can also span multiple lines using triple quotes.	"Hello, World!", 'Python Programming', """This is a multi-line string."""
bool	Represents logical values: True or False.	True, False
None	Represents the absence of a value.	None

Example:

```
x = 42       # int
pi = 3.14    # float
msg = "Merhaba" # string
a = True     # boolean
b = None     # Hiçbir nesne
```

3. Operators

3.1 Arithmetic Operators

Arithmetic operators are symbols used to perform basic mathematical operations on numbers. The most common arithmetic operators are:

Symbol	Operation	Example
+	Addition	7 + 3 = 10
-	Subtraction	10 - 4 = 6
*	Multiplication	5 * 6 = 30
/	Division	15 / 3 = 5.0 (Floating-point division)
%	Modulus	15 % 3 = 0 (Remainder 0)
//	Integer Division	15 // 3 = 5 (Integer result)
**	Exponentiation	3 ** 4 = 81 (3 to the power of 4)

3.2 Comparison Operators

Comparison operators are symbols used to compare two expressions and determine the relationship between them. The most common comparison operators are:

Symbol	Operation	Example	Result
<	Less than	x < y	x is less than y
>	Greater than	x > y	x is greater than y
==	Equal to	x == 5	x is equal to 5
!=	Not equal to	x != y	x is not equal to 5
<=	Less than or equal to	x <= y	x is less than or equal to y
>=	Greater than or equal to	x >= y	x is greater than or equal to y

Examples:

```
x = 5
y = 7

x < y  # True
x > y  # False
x == 5  # True
x != y  # True
```

3.3 Logical Operators

Logical operators are symbols used to combine two or more logical expressions and produce a new logical value. The most common logical operators are:

Symbol	Operation	Example	Result
and	And	x and y	Both x and y are true
or	Or	x or y	Either x or y is true
not	Not	not x	x is false

3.4 Bitwise Operators

Bitwise operators are symbols used to perform bit-level operations on the bits of binary numbers. The most common bitwise operators are:

Symbol	Operation	Description
&	Bitwise AND	Compares each bit of two numbers. If both bits are 1, the result is 1. Otherwise, the result is 0.
|	Bitwise OR	Bitwise OR
^	Bitwise XOR	Compares each bit of two numbers. If the two bits are different, the result is 1. Otherwise, the result is 0.
~	Bitwise NOT	Inverts each bit of a number. 1 becomes 0, and 0 becomes 1.
<<	Left Shift	Shifts the bits of a number to the left by the specified number. Shifted bits are filled with zeros.
>>	Right Shift	Shifts the bits of a number to the right by the specified number. Shifted bits are lost.

Example:

```
x = 0b1010  # Binary: 10
y = 0b1100  # Binary: 12

x & y     # 0b1000 - Sonuç: 8
x | y     # 0b1110 - Sonuç: 14
x ^ y     # 0b0110 - Sonuç: 6
~x        # 0b0101 - Sonuç: 5 (Bir'in Tersi: -(x+1) = -(10+1) = -11 = 0b....0101)
x << 2    # 0b10100 - Sonuç: 20
x >> 1    # 0b0101 - Sonuç: 5
```

3.5 Assignment Operators

Assignment operators are symbols used to assign values to variables. They can also be combined with arithmetic or bitwise operations to perform calculations and assign the result to a variable. The most common assignment operators are:

Symbol	Operation	Description	Example	Result
=	Value assignment	Assigns a value to a variable.	x = 5	x becomes 5
+=	Addition assignment	Adds a value to the existing value of a variable and assigns the result to the variable.	x += 3	x becomes 8 (x was 5 initially, 3 is added, and the result is assigned back to x)
-=	Subtraction assignment	Subtracts a value from the existing value of a variable and assigns the result to the variable.	x -= 2	x becomes 6 (x was 8 initially, 2 is subtracted, and the result is assigned back to x)
*=	Multiplication assignment	Multiplies the existing value of a variable by a value and assigns the result to the variable.	x *= 3	x becomes 18 (x was 6 initially, 3 is multiplied, and the result is assigned back to x)
/=	Division assignment	Divides the existing value of a variable by a value and assigns the result to the variable.	x /= 2	x becomes 9 (x was 18 initially, 2 is divided, and the result is assigned back to x)
%=	Modulus assignment	Performs modulus division (remainder) on the existing value of a variable and a value and assigns the result to the variable.	x %= 5	x becomes 4 (x was 9 initially, 5 is used for modulus division, and the remainder 4 is assigned back to x)
//=	Integer division assignment	Performs integer division (division without decimals) on the existing value of a variable and a value and assigns the result to the variable.	x //= 2	x becomes 2 (x was 4 initially, 2 is used for integer division, and the quotient 2 is assigned back to x)
**=	Exponentiation assignment	Raises the existing value of a variable to a power and assigns the result to the variable.	x **= 3	x becomes 64 (x was 2 initially, 3 is used for exponentiation, and the result 64 is assigned back to x)
&=	Bitwise AND assignment	Performs a bitwise AND operation on the existing value of a variable and a value and assigns the result to the variable.	x &= 7	x becomes 2 (x was 64 initially, 7 is used for bitwise AND, and the result 2 is assigned back to x)
|=	Bitwise OR assignment	Performs a bitwise OR operation on the existing value of a variable and a value and assigns the result to the variable.	x |= y	x becomes 2 (x was 64 initially, 7 is used for bitwise OR, and the result 2 is assigned back to x)
^=	Bitwise XOR assignment	Performs a bitwise XOR operation on the existing value of a variable and a value and assigns the result to the variable.	x ^= 3	x becomes 11 (x was 14 initially, 3 is used for bitwise XOR, and the result 11 is assigned back to x)
<<=	Left shift assignment	Shifts the bits of the existing value of a variable to the left by the specified number and assigns the result to the variable.	x <<= 2	x becomes 44 (x was 11 initially, 2 is used for left shift, and the result 44 is assigned back to x)
>>=	Right shift assignment	Shifts the bits of the existing value of a variable to the right by the specified number and assigns the result to the variable.	x >>= 1	x becomes 22 (x was 44 initially, 1 is used for right shift, and the result 22 is assigned back to x)

Example:

```
x = 5
x += 3   # x = 8
x *= 2   # x = 16
x %= 7   # x = 2
```

4. Control Flow

4.1. if Statements

if statements are used to execute specific code blocks based on a condition.

Example:

```
x = 5

if x < 0:
   print("Negative")
elif x == 0:
   print("Zero”)
```

4.2. while Loops

while loops repeatedly execute a block of code as long as a certain condition remains true.

Example:

```
x = 0
while x < 5:
   print(x)
   x += 1
```

4.3. for Loops

4.4. break and continue Statements

- break and continue statements are used to control the flow of loops in Python.
- break allows you to exit a loop prematurely, even if the loop condition is still true.
- continue skips the current iteration of the loop and moves on to the next one.

Example:

```
x = 0
while True:
    x += 1
    if x > 10:
        break
    if x % 2 == 0:
        continue
    print(x)
```

4.5. pass

This can be used in situations where you do not want any operation to be performed on that line.

Example:

```
def func():
    if True:
        pass  # Code will be added here later
    else:
        # ...

func()
```

5. Functions

In Pycimen, functions are defined using the def keyword. The function name is followed by parentheses containing the function parameters. The function body is separated by a double colon (:) and consists of a code block.

Example:

```
def add(a, b):
    """
    This function adds two numbers.
    """
    return a + b

total = add(3, 5)
print(total)  # Output: 8
```

Function Parameters:

Function parameters are defined as identifiers separated by commas within parentheses:

Example:

```
def function(param1, param2, param3):
    # code block
```

5.1. return Statement

The return statement is used to **return values from functions** in Python. When a function is called, the value specified in the return statement is assigned to the function.

Example:

```
def square(x):
    """
    Calculates the square of a number.
    """
    return x * x

result = square(5)
print(result)  # Output: 25
```

Without return Statement:

If a function does not contain a return statement, the function automatically returns the None value**.** This means that the function does not produce any value.

Example:

```
def greet():

    print("Hello!")

message = greet()

print(message)  # Output: None
```

5.2. Nested Function Definitions

In Pycimen, functions can be defined **inside** other functions. This allows you to write more complex and modular code.

Example:

```
def cube(x):
    """
    Calculates the cube of a number.
    """
    def square(y):
        """
        Calculates the square of a number.
        """
        return y * y
    return square(x) * x

result = cube(3)
print(result)  # Output: 27
```

6. Classes

In Pycimen, classes are defined using the class keyword. The class body is separated by a double colon (:) and defined with a code block.

Example:

```
class Car:
    """Car class"""

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        print(f"{self.brand} {self.model}")

car1 = Car("Toyota", "Corolla")
car1.display_info()  # Output: Toyota Corolla
```

Note:The special method __init__() within class definitions is automatically called when an object is created. This method is used to initialize the attributes of the class.