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.

Hint:

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.

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

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

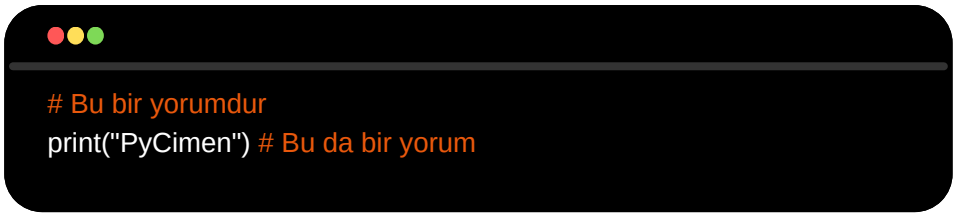
1.2. Line Breaks

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.

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

1.3. Comment Lines

Single-line comments start with the # character.



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

1.4. Multiline Comments/Docstrings

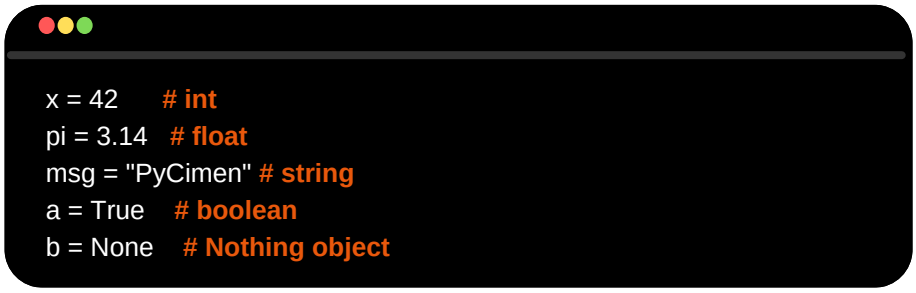
Multiline comments or docstrings are enclosed in triple quotes (""" or ```).

```
"""
This is a
multiline
docstring.
"""
```

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



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

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	Compares each bit of two numbers. If both bits are 0, the result is 0. Otherwise, the result is 1.
^	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.

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

x & y       # 0b1000 - Result: 8
x | y       # 0b1110 - Result: 14
x ^ y       # 0b0110 - Result: 6
~x          # 0b0101 - Result: 5 (One Inverse: -(x+1) = -(10+1) = -11 = 0b....0101)
x << 2      # 0b10100 - Result: 20
x >> 1      # 0b0101 - Result: 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
=	Value assignment	Assigns a value to a variable.	x = 5
+=	Addition assignment	Adds a value to the existing value of a variable and assigns the result to the variable.	x += 3
-=	Subtraction assignment	Subtracts a value from the existing value of a variable and assigns the result to the variable.	x -= 2
*=	Multiplication assignment	Multiplies the existing value of a variable by a value and assigns the result to the variable.	x *= 3
/=	Division assignment	Divides the existing value of a variable by a value and assigns the result to the variable.	x /= 2
%=	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
//=	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
**=	Exponentiation assignment	Raises the existing value of a variable to a power and assigns the result to the variable.	x **= 3
&=	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
|=	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
^=	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
<<=	Left shift assignment	Performs a bitwise XOR operation on the existing value of a variable and a value and assigns the result to the variable.	x <<= 2
>>=	Right 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 >>= 1

4. Control Flow

4.1. if Statements

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

```
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.

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

4.3. for Loops

For loops are used to iterate over iterable data structures such as lists, arrays, ranges, and strings. The for loop executes the code block in its body for each item in the iterable. You can use the loop by making it an in method.

```
e = [1, 2, 3, 4]

for val in e:
    print(val)
```

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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

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.

```
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
```

7. Module Import

In Pycimen, functions or classes from other modules can be imported using the import statement. This facilitates code reuse and modularity.This extends to a wide array of domains, ranging from game development to artificial intelligence and data analysis. With this capability, practitioners can harness the full spectrum of libraries and tools available in our language, essentially bringing all the functionalities we commonly associate with Python into our ecosystem.

Moreover, by incorporating these functionalities while benefiting from the speed and efficiency characteristic of C++, Pycimen transcends the performance limitations often associated with Python. This integration of versatility and speed empowers developers to craft solutions that are not only comprehensive but also optimized for efficiency, facilitating the creation of professional-grade applications across various domains.

Note: In Pycimen, user-defined modules can be imported in addition to standard modules. The module name should be used without the file extension.

```
import numpy
import pandas

a = numpy.random.randn(6, 4)
print(a)

b= a.mean()
print("Mean")
print(b)

data = [60, 58, 42, 55]

df = pandas.DataFrame(data)
print("First 5 rows:")
print(df.head())

print("Basic statistics:")
print(df.describe())

print("Column means:")
print(df.mean())

print("Column standard deviations:")
print(df.std())
```