COMP 1023 Introduction to Python Programming

Python Basic Operations

Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

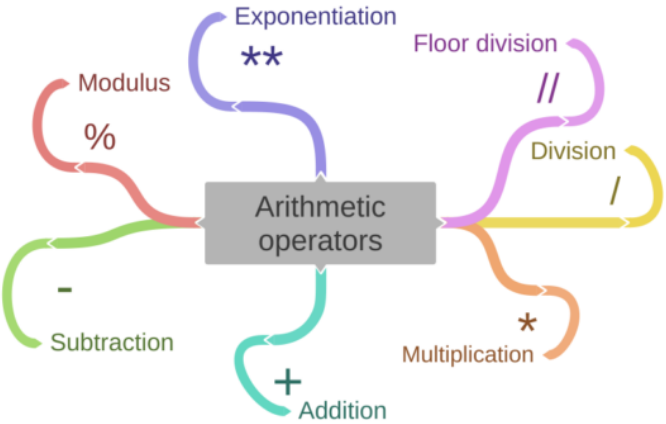
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Introduction

- According to what we covered in the last topic, now you should know the following:
  - The basic structure of Python programs.
  - Python programs can "memorize" data through objects referenced by variables.
  - Using input and print functions, we can read data from and write data to I/O devices (e.g., keyboard, monitor, etc.).
- In this topic, we will continue to discuss more about Python programming, particularly the basic operations that Python programs can perform.
- Topics to be discussed include:
  1. Arithmetic operations (also known as mathematical operations).
  2. Relational (Comparison) operations.
  3. Logical operations.
  4. Other operations.

Part I

Arithmetic Operations



Arithmetic Operations

- In Python, the following arithmetic operations are supported:

Symbol(s)	Meaning	Example	Result
`+`	Addition	`34 + 1`	`35`
`-`	Subtraction	`34.0 - 0.1`	`33.9`
`*`	Multiplication	`300 * 30`	`9000`
`/`	Float division	`1 / 2`	`0.5`
`//`	Integer division	`1 // 2`	`0`
`**`	Exponentiation	`4 ** 0.5`	`2.0`
`%`	Remainder	`20 % 3`	`2`

- `+`, `-`, `*`, `/`, `//`, `**`, `%` are called arithmetic operators.
- The input (or arguments) around the operators are called arithmetic operands.

Note

To improve readability, Python allows us to use underscores to separate digits in a number literal. For example:

```
value = 232_45_4519
amount = 23.24_4545_4519_3415
```

However, `45_` or `_45` is incorrect. The underscore must be placed between two digits.

What Can You Conclude From These?

Operator(s)	Example	Result
+	`1 + 2`	`3`
	`1 + 2.0`	`3.0`
	`1.0 + 2`	`3.0`
	`1.0 + 2.0`	`3.0`
-	`1 - 2`	`-1`
	`1 - 2.0`	`-1.0`
	`1.0 - 2`	`-1.0`
	`1.0 - 2.0`	`-1.0`
*	`1 * 2`	`2`
	`1 * 2.0`	`2.0`
	`1.0 * 2`	`2.0`
	`1.0 * 2.0`	`2.0`
/	`1 / 2`	`0.5`
	`1 / 2.0`	`0.5`
	`1.0 / 2`	`0.5`
	`1.0 / 2.0`	`0.5`

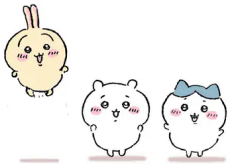
- In Python, if one of the operands for `+`, `-`, `*` is of float type, the result of the operation will also be float.
  In fact, this also applies to `//`, `**`, and `%`, which we will introduce later.
- For `/`, the result will always be float.

Note

If an integer and a float are involved in a binary operation, Python automatically converts the integer to a float value. This is called implicit type conversion.

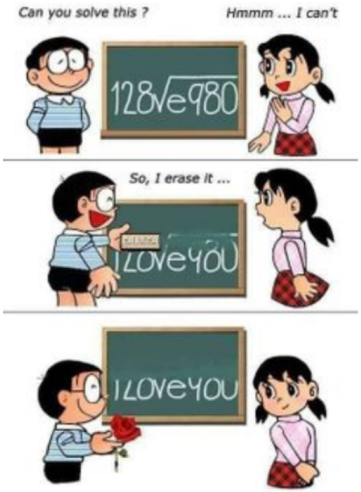
Arithmetic Operations

```
def main():  # Filename: arithmetic_operations.py
    # Addition
    print('1 + 2 =', 1 + 2, 'and type is', type(1 + 2))
    print('1 + 2.0 =', 1 + 2.0, 'and type is', type(1 + 2.0))
    print('1.0 + 2 =', 1.0 + 2, 'and type is', type(1.0 + 2))
    print('1.0 + 2.0 =', 1.0 + 2.0, 'and type is', type(1.0 + 2.0))
    # Subtraction
    print('1 - 2 =', 1 - 2, 'and type is', type(1 - 2))
    print('1 - 2.0 =', 1 - 2.0, 'and type is', type(1 - 2.0))
    print('1.0 - 2 =', 1.0 - 2, 'and type is', type(1.0 - 2))
    print('1.0 - 2.0 =', 1.0 - 2.0, 'and type is', type(1.0 - 2.0))
    # Multiplication
    print('1 * 2 =', 1 * 2, 'and type is', type(1 * 2))
    print('1 * 2.0 =', 1 * 2.0, 'and type is', type(1 * 2.0))
    print('1.0 * 2 =', 1.0 * 2, 'and type is', type(1.0 * 2))
    print('1.0 * 2.0 =', 1.0 * 2.0, 'and type is', type(1.0 * 2.0))
    # Division
    print('1 / 2 =', 1 / 2, 'and type is', type(1 / 2))
    print('1 / 2.0 =', 1 / 2.0, 'and type is', type(1 / 2.0))
    print('1.0 / 2 =', 1.0 / 2, 'and type is', type(1.0 / 2))
    print('1.0 / 2.0 =', 1.0 / 2.0, 'and type is', type(1.0 / 2.0))

if __name__ == "__main__":
    main()
```



Output of Arithmetic Operations

```
1 + 2 = 3 and type is <class 'int'>
1 + 2.0 = 3.0 and type is <class 'float'>
1.0 + 2 = 3.0 and type is <class 'float'>
1.0 + 2.0 = 3.0 and type is <class 'float'>

1 - 2 = -1 and type is <class 'int'>
1 - 2.0 = -1.0 and type is <class 'float'>
1.0 - 2 = -1.0 and type is <class 'float'>
1.0 - 2.0 = -1.0 and type is <class 'float'>

1 * 2 = 2 and type is <class 'int'>
1 * 2.0 = 2.0 and type is <class 'float'>
1.0 * 2 = 2.0 and type is <class 'float'>
1.0 * 2.0 = 2.0 and type is <class 'float'>

1 / 2 = 0.5 and type is <class 'float'>
1 / 2.0 = 0.5 and type is <class 'float'>
1.0 / 2 = 0.5 and type is <class 'float'>
1.0 / 2.0 = 0.5 and type is <class 'float'>
```

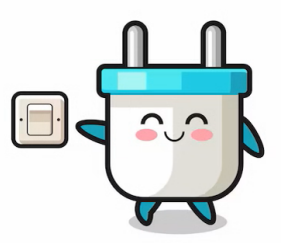


Integer Division (//)

- The operator `//` is used for integer division, also known as floor division.
- The `//` operator divides two numbers and returns the largest integer that is less than or equal to the result (the floor of the division).
- Examples:
  - `5 // 2` returns `2`, because 5 divided by 2 is 2.5, and the largest integer less than or equal to 2.5 is 2.
  - `5.0 // 2` returns `2.0`, because the result is a float.
  - `5 // 2.0` also returns `2.0`.
  - `5.5 // 2` returns `2.0`, as the largest integer less than or equal to 2.75 is 2.
  - `5.5 // -2` returns `-3.0`, as the largest integer less than or equal to -2.75 is -3.

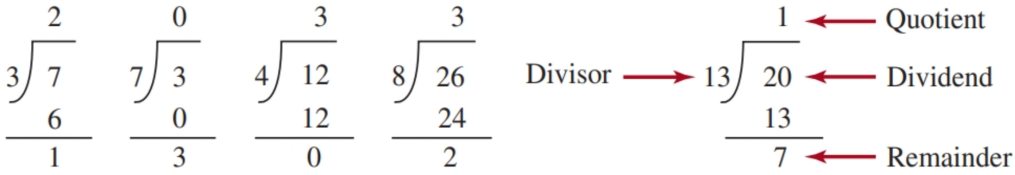
Exponentiation (**)

- The operator `**` is used for exponentiation, which raises a number (the base) to the power of another number (the exponent).
- Examples:
  - `2 ** 3` evaluates to `8`, because it is equivalent to $2^3 = 8$.
  - `4.0 ** 0.5` evaluates to `2.0`, because it is equivalent to $4.0^{0.5} = 4.0^{\frac{1}{2}} = \sqrt{4.0} = 2.0$.
  - `2 ** -2` evaluates to `0.25`, because it is equivalent to $2^{-2} = \frac{1}{2^2} = 0.25$.
  - `2 ** 3 ** 2` evaluates to `512`, because it is equivalent to $2^{(3**2)}$, which is 2^9 and equals 512.



Modulo Operation (%)

- The arithmetic operator % is referred to as the "modulo operator", "mod operator", or simply "mod".
- It is used to find the remainder after integer division.



- The modulo operator has many useful applications in computer programs:
  - Testing whether a number is even or odd: `number % 2 == 0` for even numbers, `number % 2 == 1` for odd numbers.
  - Finding individual digits of a number: `number % 10` gives the least significant digit (rightmost digit) of the number.
  - Finding the last four digits of an HKID number: `ID % 10000`.

Modulo Operations

```python
# Filename: mod_operations.py
# An example demonstrating mod operations
def main():
    print('1023 % 2 is', 1023 % 2)       # Remainder when 1023 is divided by 2
    print('1023 % 10 is', 1023 % 10)     # Remainder when 1023 is divided by 10
    print('1023 % 100 is', 1023 % 100)   # Remainder when 1023 is divided by 100
    print('1023 % 1000 is', 1023 % 1000) # Remainder when 1023 is divided by 1000
    # Last 4 digits of ID
    print('The last 4 digits of ID: 8121232 % 10000 is', 8121232 % 10000)

if __name__ == "__main__":
    main()
```

Output:

```
The result of 1023 % 2 is 1
The result of 1023 % 10 is 3
The result of 1023 % 100 is 23
The result of 1023 % 1000 is 23
The last 4 digits of ID: 8121232 % 10000 is 1232
```

Display Time

- This program converts an amount of time given in seconds into minutes and remaining seconds.

```
# Filename: display_time.py
# A program that obtains minutes and remaining seconds from an amount of time in seconds
def main():
    # Prompt the user for input
    seconds = int(input("Enter an integer for seconds: "))

    # Calculate minutes and remaining seconds
    minutes = seconds // 60  # Calculate whole minutes
    remaining_seconds = seconds % 60  # Calculate remaining seconds

    # Display the result
    print(seconds, "seconds is", minutes, "minutes and", remaining_seconds, "seconds")

if __name__ == "__main__":
    main()
```

Output:

```
Enter an integer for seconds: 500
500 seconds is 8 minutes and 20 seconds
```

More about Modulo Operations

- The operands of the modulo operator can be positive or negative integers or floats.
- For `a % b`, it is computed based on the following formula:

$$r = a - (a//b) * b$$

- Examples:
  - `10 % 3` $=$ `10 - (10 // 3) * 3` $=$ `10 - 3 * 3` $=$ `1`
  - `10 % -3` $=$ `10 - (10 // -3) * (-3)` $=$ `10 - (-4) * (-3)` $=$ `10 - 12` $=$ `-2`
  - `-10 % 3` $=$ `-10 - (-10 // 3) * 3` $=$ `-10 - (-4) * 3` $=$ `-10 + 12` $=$ `2`
  - `-10 % -3` $=$ `-10 - (-10 // -3) * (-3)` $=$ `-10 - 3 * (-3)` $=$ `-10 + 9` $=$ `-1`
  - `-5.3 % 3` $=$ `-5.3 - (-5.3 // 3) * 3` $=$ `-5.3 - (-2) * 3` $=$ `-5.3 + 6` $=$ `0.7`
  - `5.3 % -3` $=$ `5.3 - (5.3 // -3) * (-3)` $=$ `5.3 - (-2) * (-3)` $=$ `5.3 - 6` $=$ `-0.7`
  - `-5.3 % -3` $=$ `-5.3 - (-5.3 // -3) * (-3)` $=$ `-5.3 - 1 * (-3)` $=$ `-5.3 + 3` $=$ `-2.3`

Can you calculate `6.4 % -1.7`?

Subtraction and Negation Operations (-)

- The operator - is used as both a binary subtraction operator and a unary negation operator.
  - The binary subtraction operator takes 2 operands (e.g., `10 - 20`).
  - The unary negation operator takes 1 operand (e.g., `-3`).

```
# Filename: subtraction_negation.py

def main():
    print("The result of 10 - 20 is", 10 - 20) # Subtraction
    print("This is", -3)                       # Negation

if __name__ == "__main__":
    main()
```

Output:

```
The result of 10 - 20 is -10
This is -3
```

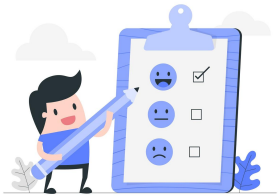
Expressions

- In programming, we often need to include values and operations; the technical term for these is expressions.
- An expression is a simple value or a combination of operations that produces a value.
- Examples:
  - Simplest expressions (just an integer or a floating-point literal): `89` or `30.23`.
  - A combination of operations that produces a value:

    ```
    (3 * 7) + (4 * 4) + 8
    ```

    (The resulting value of the above expression is `45`).

The process of obtaining the value of an expression is called evaluation.

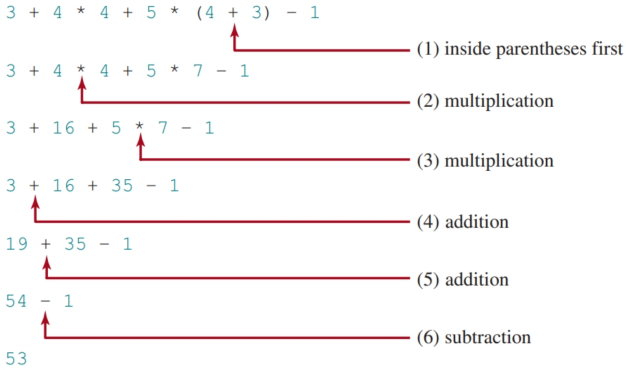


Precedence of Arithmetic Operators

Q: If there are multiple operators involved in an expression, how should we evaluate it?

- This is determined by the precedence of operators.
- Operator precedence determines the order in which operations are performed in an expression.
- For arithmetic expressions, there are three levels of precedence:
  - Exponentiation (`**`) is applied first.
  - Multiplication (`*`), float division (`/`), integer division (`//`), and remainder (`%`) operators are applied next. If an expression contains several of these operations, they are applied from left to right.
  - Addition (`+`) and subtraction (`-`) operators are applied last. If an expression contains multiple addition and subtraction operators, they are also applied from left to right.
- Can I override these rules?
  Yes, by using parentheses (`()`).

Precedence of Arithmetic Operators



- This diagram illustrates the precedence of arithmetic operators in Python.
- Higher precedence operators are evaluated before lower precedence operators.
- Use parentheses to override the default precedence rules.

Precedence of Arithmetic Operators

```
# Filename: precedence_arithmetic_operators.py

def main():
    print("The result of 3 + 4 * 4 + 5 * (4 + 3) - 1 is",
          3 + 4 * 4 + 5 * (4 + 3) - 1)

if __name__ == "__main__":
    main()
```

Output:

```
The result of 3 + 4 * 4 + 5 * (4 + 3) - 1 is 53
```

Arithmetic Operations on Variables

- Arithmetic operators can also be applied to variables.
- The precedence of operations remains the same when using variables.
- This example demonstrates various arithmetic operations on variables:

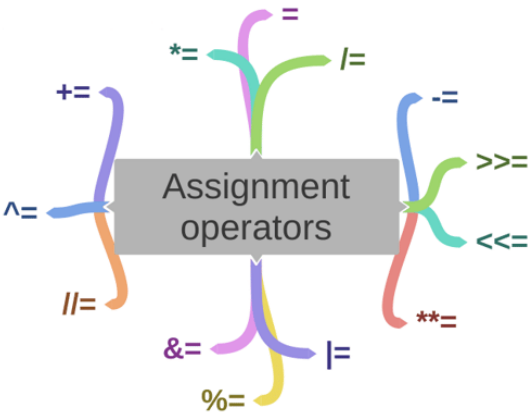
```
def main():
    a, c = 10, 22
    b, d = 20.1, 18.9
    print("Addition:", a + b)
    print("Subtraction:", a - b)
    print("Multiplication:", a * b)
    print("Division:", a / b)
    # The following has a precision error
    print("Combination:", a * 10 + b / c - d + 6)

if __name__ == "__main__":
    main()
```

Output:

```
Addition: 30.1
Subtraction: -10.100000000000001
Multiplication: 201.0
Division: 0.49751243781094523
Combination: 88.01363636363635
```

Part II

Assignment Operation



Assignment Operation

- The assignment operator = is used to assign a value to a variable that references an object.

Syntax

```
<var name> = <initial value>
# Cascading assignment
<var name1> = <var name2> = <initial value>
# Simultaneous assignments
<var name1>, <var name2>, ... = <initial value1>, <initial value2>, ...
<var name> = <expression>
<var name1> = <var name2> = <expression>
<var name1>, <var name2>, ... = <expression1>, <expression2>, ...
```

where <var name> or <var nameN> is the variable name in the program, referring to its location in memory; <initial value> or <initial valueN> is the value initially assigned to the object referenced by a variable (this is called initialization); <expression> or <expressionN> refers to a value obtained from operations.

Assignment Operation

- Examples:

```
# Simultaneous assignments
x, y = 3, 4   # Create objects with values 3 and 4 and referenced by x and y
x = 2 + 9     # Create an object with the result of 2 + 9 (11) and referenced by x
x = y         # Create an object to store the value of y (4) and referenced by x
x = x + 1     # Create an object to store the value of x + 1 (5) and referenced by x
```

- Common Confusion:
  - Many students may get confused because assignment statements, such as `x = x + 1`, do not follow traditional mathematical rules.
  - In mathematics, no real number can satisfy the equation `x = x + 1`.

  However, in programming, we are simply assigning the value of `(x + 1)` to the variable `x`. This is not a mathematical equation, but an assignment operation.

Assignment Operation

```python
# Filename: bmi_example.py

def main():
    weight, height = 195, 70  # Initialize weight and height
    bmi = weight / (height * height) * 703  # Calculate BMI
    print("Previous BMI:", bmi)

    weight = 180  # Update weight
    bmi = weight / (height * height) * 703  # Recalculate BMI
    print("Current BMI:", bmi)

if __name__ == "__main__":
    main()
```

Output:

```
Previous BMI: 27.976530612244897
Current BMI: 25.82448979591837
```

What is Wrong in This Code?

```
def main():
    count = count + 1  # Error: count is not defined yet
    print(count)

if __name__ == "__main__":
    main()
```

- An error occurs in the line `count = count + 1` because the variable `count` has not been defined before this line.
- To fix it, we can define the object referenced by the variable `count` first:

```
def main():
    count = 1          # Initialize count
    count = count + 1  # Increment count by 1
    print(count)

if __name__ == "__main__":
    main()
```

Simultaneous Assignments and Cascading Assignments

```python
# Filename: simultaneous_cascading_assignment.py

def main():
    x, y = 10, 20  # Simultaneous assignments
    ''' This is equivalent to:
        x = 10
        y = 20 '''
    print(x, y)

    i = j = k = 1  # Cascading assignments
    ''' This is equivalent to:
        k = 1
        j = k
        i = j '''
    print(i, j, k)

if __name__ == "__main__":
    main()
```

Output:

```
10 20
1 1 1
```

Assignment Operation - Swapping Two Variables

```python
# Filename: swapping.py
def main():
    x = 5
    y = 10

    # Swap x with y
    x, y = y, x
    print(f"x: {x}, y: {y}")

    ''' x, y = y, x
    is equivalent to:
        temp = x
        x = y
        y = temp '''

if __name__ == "__main__":
    main()
```

How it Works?

We will discuss this in detail when we talk about tuples!

- Tuple Packing: On the right side of the assignment `(y, x)`, Python creates a tuple containing the current values of `y` and `x`. For example, if `x` is 5 and `y` is 10, the expression `(y, x)` evaluates to the tuple `(10, 5)`.
- Tuple Unpacking: On the left side of the assignment, the values from this tuple are unpacked into the variables `x` and `y`. In this case, `x` will get the value of 10, and `y` will get the value of 5.

Augmented Assignment Operations

- Python allows augmented assignment operations to simplify coding.

Operator	Description	Example	Equivalent
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Float division assignment	`i /= 8`	`i = i / 8`
`//=`	Integer division assignment	`i //= 8`	`i = i // 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`
`**=`	Exponentiation assignment	`i **= 8`	`i = i ** 8`

These augmented assignment operators are syntactic sugar, providing more convenient and expressive ways to perform operations that could be done with more verbose syntax.

- Example:
  - `x /= 4 + 5.5 + 1.5` is equivalent to `x = x / (4 + 5.5 + 1.5)`.

Note

There are no spaces in augmented assignment operators. For example, `+=` should not have a space.

Augmented Assignment Operation

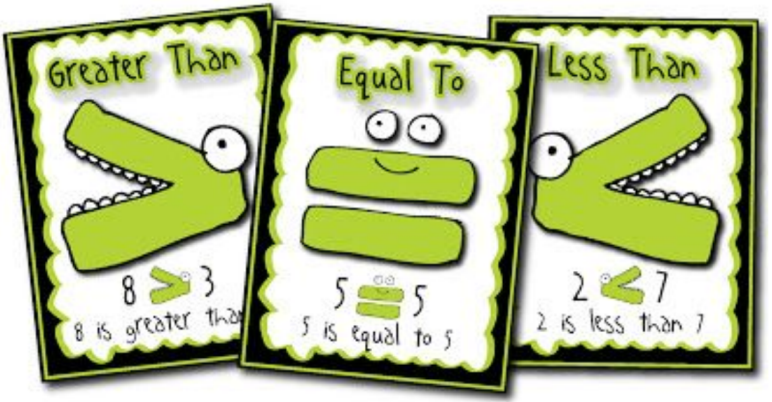
```
# Filename: augmented_assignment.py

def main():
    account = 2000.0
    income = 1000.0
    print("Account =", account, ", income =", income)
    account += income  # Augmented assignment to update account
    print("Account =", account, ", income =", income)

if __name__ == "__main__":
    main()
```

Output:

```
Account = 2000.0 , income = 1000.0
Account = 3000.0 , income = 1000.0
```

Part III

Relational (Comparison) Operations



Relational Operations

- Relational (comparison) operators define the relationships between values.

Operator	Maths Symbol	Description	Example (radius is 5)	Result
<	$<$	less than	radius < 0	False
<=	$\leq$	less than or equal to	radius <= 0	False
>	$>$	greater than	radius > 0	True
>=	$\geq$	greater than or equal to	radius >= 0	True
==	$=$	equal to	radius == 0	False
!=	$\neq$	not equal to	radius != 0	True

- The result of the comparison is a Boolean value: True or False.

```
radius = 5
print(radius > 0)  # Prints True
```

Relational Operations

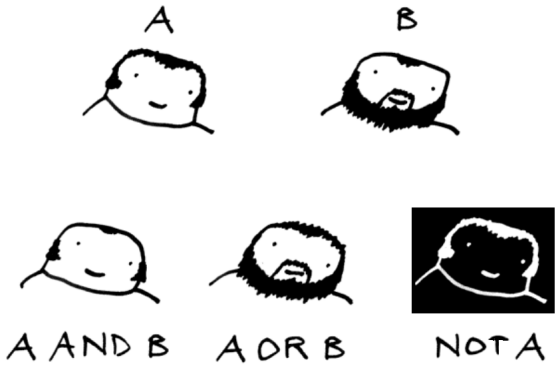
```python
def main():  # Filename: relational_operations.py
    x = 10
    y = 20
    x_greater_than_y = (x > y)
    x_greater_than_equal_y = (x >= y)
    x_less_than_y = (x < y)
    x_less_than_equal_y = (x <= y)
    x_equal_y = (x == y)
    x_not_equal_y = (x != y)

    print("x, y:", x, y)
    print("x > y:", x_greater_than_y)
    print("x >= y:", x_greater_than_equal_y)
    print("x < y:", x_less_than_y)
    print("x <= y:", x_less_than_equal_y)
    print("x == y:", x_equal_y)
    print("x != y:", x_not_equal_y)

if __name__ == "__main__":
    main()
```

Output:

```
x, y: 10 20
x > y: False
x >= y: False
x < y: True
x <= y: True
x == y: False
x != y: True
```

Part IV

Logical Operations



Logical Operations

- There are three logical operators in Python:

Operator	Description
`not`	logical negation
`and`	logical conjunction
`or`	logical disjunction

- Logical operators combine two Boolean values and produce a single Boolean result based on the corresponding truth table.

Truth Table				
p	**q**	**p and q**	**p or q**	**not p**
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

- The variables p and q can be Boolean literal values or evaluated Boolean values from expressions.

Logical Operations

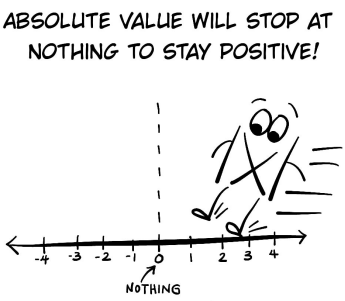
- Return True if x is in the range $[3, 10]$:

  ```
  in_range = ( (x >= 3) and (x <= 10) )
  ```

  Python supports chained comparison operators:

  ```
  # in_range = ( 3 <= x <= 10 ) # Equivalent statement
  ```

- Return True if $|x| > 1$:
  - `abs_x_greater_than_1 = ( (x > 1) or (x < -1) )`
- Return True if x is not equal to 0. The two statements below are equivalent:
  - `non_zero = (x != 0)`
  - `non_zero = not(x == 0)`



Example: Determining Leap Years

- A leap year has 366 days. The February of a leap year has 29 days.
- A leap year is divisible by 4 but not by 100, or divisible by 400

```
def main():  # Filename: leap_year.py
    year = int(input("Enter a year: "))

    # Check if the year is a leap year
    is_leap_year = (year % 4 == 0 and year % 100 != 0) or \
                   (year % 400 == 0)

    # Display the result
    print(year, "is a leap year?", is_leap_year)

if __name__ == "__main__":
    main()
```

Output:

```
Enter a year: 2000
2000 is a leap year? True
```

Short Circuiting or Lazy Operations

- Short-circuiting is a technique where the execution of a boolean expression containing 'or' and 'and' operators is halted as soon as the truth value of the expression is determined.
- This technique makes computation faster by not evaluating the remaining expressions once the truth value is established.

Operation	Notes
`p or q`	q is executed only if p is False
`p and q`	q is executed only if p is True

- The evaluation of expressions takes place from left to right.

```
print(True or 2/0)      # Evaluates to True, 2/0 is not evaluated
print(False or 2/0)     # This raises an error, since 2/0 is evaluated

print(False and 2/0)    # Evaluates to False, 2/0 is not evaluated
print(True and 2/0)     # This raises an error, since 2/0 is evaluated
```

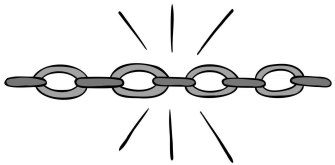
Chain Operators in Python

- Chain operators allow you to combine multiple comparisons in a single expression.
- They evaluate to 'True' if all comparisons are 'True'.
- Example:

$$5 > x > 3$$

This checks if x is greater than 3 and less than 5.

The expression `5 > x > 3` is equivalent to `x < 5 and x > 3`.



Part V

Other Operations

Identity Operations

- Identity operators check if two variables refer to the same object in memory.
- There are two identity operators:
  - `is`: Returns `True` if both operands refer to the same object.
  - `is not`: Returns `True` if both operands do not refer to the same object.
- Use identity operators when you need to check if two variables point to the same object, not just if they are equal in value.

```
a = 1
b = 1
print(a is b)       # Print True
print(a is not b)   # Print False
```

Since both a and b are assigned the value 1, and Python caches small integers (typically between -5 and 256), both a and b refer to the same integer object in memory. Therefore, `print(a is b)` evaluates to `True`, and `print(a is not b)` evaluates to `False`.

Python Caches Small Integers

- Python caches small integers for efficiency, typically in the range of −5 to 256. But the range of these cached integers can vary depending on the platform and the Python implementation.
- This means that when you create an integer within this range, Python reuses the same object instead of creating a new one.

```
x = 100; y = 100
print(x is y)  # Print True
```

  - The `is` operator confirms that both `x` and `y` point to the same object in memory.
  - This caching mechanism can lead to performance improvements and reduced memory usage.
  - However, integers outside this range are not cached:

```
a = 1000; b = 1000
print(a is b)  # Print True in VS Code, but False in Colab
```

  - In Colab, `a` and `b` are different objects in memory.

Caching small integers is an optimization technique employed by Python to enhance performance for frequently used values.

Membership Operations

- Membership operators check if an item exists within a collection. They return a boolean value `True` or `False`.
- There are two membership operators:
  - `in`: Returns `True` if the specified item exists in a sequence; `False` otherwise.
  - `not in`: Returns `True` if the specified item does not exist in a sequence; `False` otherwise.

Note

More details will be provided when discussing collections!

Associativity of Operators

- Associativity refers to the order in which operators of the same precedence are evaluated in an expression.
- In Python, operators can be left-to-right associative or right-to-left associative.
- Left-to-right Associative Operators:
  - Most arithmetic operators (e.g., +, -, *, /) are left-to-right associative.

    `10 / 5 * 2` is evaluated as `(10 / 5) * 2 = 4`
- Right-to-left Associative Operators:
  - The exponentiation operator (**) is right-to-left associative.

    `2 ** 3 ** 2` is evaluated as `2 ** (3 ** 2) = 2 ** 9 = 512`

Operator Precedence from High (Top) to Low (Down)

Operator	Description	Associativity
`**`	Exponentiation	Right-to-left
`+, -`	Unary plus and minus	Right-to-left
`*, /, //, %`	Multiplication, division, integer division, and modulus	Left-to-right
`+, -`	Binary addition and subtraction	Left-to-right
`<, <=, >, >=`	Relational operators	Left-to-right
`==, !=`	Equality operators	Left-to-right
`is, is not`	Identity operators	Left-to-right
`in, not in`	Membership operators	Left-to-right
`not`	Logical negation	Left-to-right
`and`	Logical conjunction	Left-to-right
`or`	Logical disjunction	Left-to-right
`=, +=, -=, *=, /=, //=, %=, **=`	Assignment and augmented assignment operators	Right-to-left

Reference: `https://docs.python.org/3/reference/expressions.html`

Do not rely solely on precedence rules when writing your program.

- Be cautious: logical `and` is evaluated before logical `or`.
- Example: `(p or q and r)` is equivalent to `(p or (q and r))`. Use parentheses to clarify your intention.

Floating Point Errors in Python

- Occur due to the way floating point numbers are represented in computer memory.
- Can lead to precision loss and unexpected results in calculations.
- Common Causes:
  - Precision Limitations:
    - Floating point numbers have limited precision (typically 15-17 decimal digits).
    - Example: `0.1 + 0.2` may not equal `0.3`.
  - Rounding Errors:
    - Operations can introduce small errors due to rounding.
    - Example: `print(0.1 + 0.2 == 0.3)` results in `False`.
- Handling Floating Point Errors:
  - Use the `math.isclose()}` function for comparisons:

```
import math
print(math.isclose(0.1 + 0.2, 0.3))  # Print True
```

  - Consider using the `decimal` module for higher precision:

```
from decimal import Decimal
print(Decimal('0.1') + Decimal('0.2') == Decimal('0.3'))  # Print True
```

Common Pitfalls in Python Programming

- Confusion between Assignment and Equality:
  - The assignment operator = is used to assign values to variables.
  - The equality operator == is used to compare two values.
  - Common mistake: Using = when you meant to use ==.
- Misunderstanding Operator Precedence:
  - Operators have different levels of precedence, which affects the order of operations.
  - Example: In the expression `a + b * c`, multiplication is performed before addition.
  - Common mistake: Assuming operations will be executed from left to right without considering precedence.
- Using `is` vs. ==:
  - The `is` operator checks for identity (same object in memory), while == checks for equality (same value).
  - Common Mistake: Using `is` to compare values instead of ==.

Key Terms

- Arithmetic operators/operations
- Assignment operator/operations
- Augmented assignment operators/operations
- Cascading assignments
- Evaluations
- Expressions
- Exponentiation
- Identity operators/operations
- Integer division
- Lazy operator
- Logical operators
- Membership operators/operations
- Modulo operation
- Negation operator/operation
- Operands
- Operators
- Precedence
- Relational operators/operations
- Short-circuit evaluation
- Simultaneous assignments
- Syntactic sugar
- Truth table

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. In Python, three main types of operations are _____, _____, and _____.
2. `+`, `-`, `*`, `/`, `//`, `**`, `%` are called _____, and the arguments around them are called _____.
3. When evaluating an expression with values of an `int` type and a `float` type, Python automatically converts the _____ type value to a _____ type value.
4. `//` is used for _____.
5. `**` is used for _____.
6. `%` is called the _____ operator, and is used to find the _____ after integer division.

Answer: 1. arithmetic, relational, logical, 2. operators, operands, 3. int, float, 4. integer division, 5. exponentiation, 6. modulo, remainder.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

7. _____ is used as both a binary operator and a unary operator.
8. An _____ is a simple value or a combination of operations that produces a value.
9. _____ can be used to force the order of evaluation to occur in any sequence.
10. The _____ is used to assign a value to an object referenced by a variable.
11. The operators in arithmetic expressions are evaluated in the order determined by the rules of _____ and _____.
12. Python provides _____ operators: `+=`, `-=`, `*=`, `/=`, `//=`, and `%=`.
13. The _____ operators (`<`, `<=`, `==`, `!=`, `>`, `>=`) yield a Boolean value.
14. Python uses _____ to represent `True`, and _____ for `False`.

Answer: 7. Operator `-`, 8. expression, 9. Parenthesis, 10. assignment operator, 11. parenthesis, operator precedence, 12. augmented assignment, 13. relational (or comparison), 14. `(1)`, `(0)`.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

15. _____ values are considered `True` in a Boolean context, while _____ values are evaluated as `False`.
16. The logical operators _____, _____, and _____ operate with Boolean values and variables.
17. _____ is a technique where the execution of a boolean expression containing `or` and `and` operators is halted.
18. Logical _____ is evaluated before logical _____.
19. _____ operators check if two variables refer to the same object in the memory.
20. _____ operators check if an item exists, or does not exist within a sequence.

Answer: 15. Non-zero, zero, 16. `and`, `or`, `not`, 17. Short circuiting, 18. `and`, `or`, 19. Identity, 20. Membership

Further Reading

- Read Sections 2.5 - 2.6, 2.8 - 2.12, 3.2, 3.10, and 3.15 of “Introduction to Python Programming and Data Structures” textbook.

That's all!

Any questions?