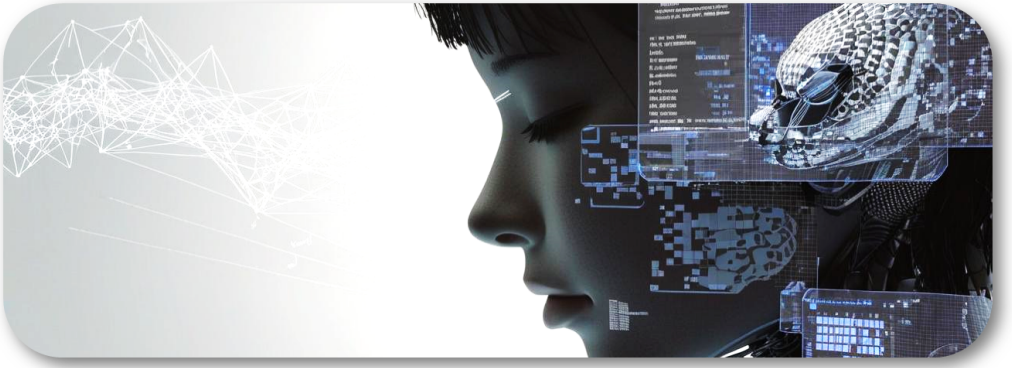


COMP 1023 Introduction to Python Programming

Tutorial 3: Operations

COMP 1023 Teaching Team

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

Representing Numbers in Computers

- Most programming languages have at least two categories of data types to represent numbers
- Integral data types, which represent integers only
- Non-integral data types, which represent decimals (includes integers)
- Computer Organization (COMP 2611 or ELEC 2350) will go into more details how numbers are represented in memory

Representing Integers in Computers

- Integral data types, which represent integers only
- In Python...
  - `int` takes a dynamic number of bytes
  - `int` can represent arbitrarily small or large integers
- In many other programming languages...
  - Often have a fixed number of bytes
  - Often can only represent a finite range of integers

Representing Decimals in Python

- Non-integral data types, which represent decimals (includes integers)
- In Python...
  - `float` usually takes 8 bytes (64 bits)
  - Its representation, IEEE 754, is similar to non-integral data types in other programming languages
- There are (uncountably) infinite real numbers, so it is impossible for `float` to represent every real number using 8 bytes only
- The nearest representable real number is used
- Precision loss in representation and calculations when using `float`!
  - Python also has a rarely used data type `decimal.Decimal`, which does not suffer from precision loss up to a user-specified precision

iPRS Question 1

What does the following code print?

```
value = float("3e14")
print("Type:", type(value), sep="\n")
```

A. `Type:`
   `<class 'float'>`

B. `Type: 300000000000000.0`

C. `ValueError: could not convert string to float: '3e14'`

D. `Type: <class 'float'>`

E. `Type: <class 'int'>`

iPRS Question 1 - Answer

What does the following code print?

```
value = float("3e14")
print("Type:", type(value), sep="\n")
```

A. `Type:`
   `<class 'float'>`

B. `Type: 300000000000000.0`

C. `ValueError: could not convert string to float: '3e14'`

D. `Type: <class 'float'>`

E. `Type: <class 'int'>`

Explanation

- 3e14 represents a number in scientific notation; scientific notation always use `float`
- The separator (`sep`) between each object provided to `print` is changed to a newline instead of a space

Representing Decimals in Python

A classical example...

```
import math

print(1 + 2) # 3
print(1 + 2 == 3) # True

print(0.1 + 0.2) # 0.30000000000000004
print(0.1 + 0.2 == 0.3) # False
print(math.isclose(0.1 + 0.2, 0.3)) # True
print(abs((0.1 + 0.2) - 0.3) < 1e-10) # Similar to `math.isclose`
# In general, use `math.isclose` to check for `float` equality.
```

Always keep this in mind when using `floats`!

Further reading: Floating-Point Arithmetic: Issues and Limitations

iPRS Question 2

What does the following code print?

```
import math
x = math.pi       # π
y = math.tan(x)   # tan(π)
print(y)          # ???
```

A. Zero

B. A nonzero value very far away from zero

C. A nonzero value very close to zero

iPRS Question 2 - Answer

What does the following code print? In mathematics, $\tan(\pi) = 0$.

```
import math
x = math.pi       # π
y = math.tan(x)   # tan(π)
print(y)          # ???
```

A. Zero

B. A nonzero value very far away from zero

C. A nonzero value very close to zero

Explanation

π requires an infinite number of digits to represent exactly, so it cannot be represented exactly by a `float`! A `float` in Python can represent 15 to 17 significant digits.

Arithmetic Operators

- Refer to lecture slides
  - Arithmetic operations
- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Floor division: `//`
- Exponentiation: `**`
- Remainder: `%`

Exercise: Arithmetic Operators

Convert the following natural language sentences into a one-line Python codes.

1. Add `x` to `y`
2. Subtract `x` from `y`
3. Multiply `x` by `y`
4. (Demo) Divide `x` by `y`
5. (Demo) `x` divides `y`
6. Remainder of dividing `x` by `y`
7. Square of `x`
8. Square root of `x`
9. `x`-th power of `y`
10. The largest integer that is not larger than the result of dividing `x` by `y`
11. (Challenge) The smallest integer that is not smaller than the result of dividing `x` by `y`

Exercise: Arithmetic Operators

Convert the following natural language sentences into a one-line Python codes.

1. Add `x` to `y`: `x + y` or `y + x` (addition is commutative)
2. Subtract `x` from `y`: `y - x`
3. Multiply `x` by `y`: `x * y` or `y * x` (multiplication is commutative)
4. Divide `x` by `y`: `x / y`
5. `x` divides `y`: `y / x`
6. Remainder of dividing `x` by `y`: `x % y`
7. Square of `x`: `x ** 2` or `x * x`
8. Square root of `x`: `x ** 0.5` or `x ** (1 / 2)`
9. `x`-th power of `y`: `y ** x`
10. The largest integer that is not larger than the result of dividing `x` by `y`: `x // y`
    - This takes the floor of the result of division, i.e. floor division
11. (Challenge) The smallest integer that is not smaller than the result of dividing `x` by `y`: `-(-x // y)`
    - This takes the ceiling of the result of division
    - Take a moment to see why the above code is correct...

Exercise: Order and Associativity of Operations

Refer to lecture slides

- Precedence of arithmetic operators
- Associativity of arithmetic operators

Evaluate the following Python arithmetic expressions. No need to care about the return type for now.

1. `1 + 2 * 3`
2. (Demo) `(1 + 2) * 3`
3. `1 + 2 + 3`
4. `1 + (2 + 3)`
5. `2 ** 2 ** 3`
6. `(2 ** 2) ** 3`
7. (Demo) `-1 ** 2`

Exercise: Order and Associativity of Operations

Evaluate the following Python arithmetic expressions. No need to care about the return type for now.

1. `1 + 2 * 3`: 7
2. `(1 + 2) * 3`: 9
3. `1 + 2 + 3`: 6
4. `1 + (2 + 3)`: 6
5. `2 ** 2 ** 3`: 256
6. `(2 ** 2) ** 3`: 64
7. `-1 ** 2`: -1
   - Exponentiation has a higher operator precedence than negation
   - Consider writing the same expression in mathematics: $-1^2 = -\left(1^2\right)$

Return Type of Arithmetic Operators

- Only consider two possible input and output types: `int`, `float`
- Not consider operations that may return other types, e.g. `complex`
- Rules of thumb to help memorize return type: see appendix

Exercise: Return Type of Arithmetic Operators

State the return type of the following expressions.

1. `1 + 2`
2. `7 / 4`
3. `7 // 4`
4. `7.  // 4`
5. `(1 + 2) * 3 * 4 * 1`
6. (Demo) `(1 + 2) * 3 * 4 / 1`
7. `(1 + 2) * 3 * 4 * 1e0`
8. `4 ** 95`
9. (Demo) `4 ** -95`

Exercise: Return Type of Arithmetic Operators

State the return type of the following expressions.

1. `1 + 2`: `int`
2. `7 / 4`: `float`
3. `7 // 4`: `int`
4. `7.  // 4`: `float`
5. `(1 + 2) * 3 * 4 * 1`: `int`
6. `(1 + 2) * 3 * 4 / 1`: `float`
   - The final operation is division (`/`), which produces a `float`
7. `(1 + 2) * 3 * 4 * 1e0`: `float`
   - Numbers in scientific notation are always `floats`
8. `4 ** 95`: `int`
9. `4 ** -95`: `float`
   - Taking the negative power requires division

iPRS Question 3

Which of the following inputs will make the following program produce a runtime error?

```
xx = float(input())
print("Reciprocal is", xx ** -1)
```

A. 4.2

B. 1

C. 0

D. -1

iPRS Question 3 - Answer

Which of the following inputs will make the following program produce a runtime error?

```
xx = float(input())
print("Reciprocal is", xx ** -1)
```

A. 4.2

B. 1

C. 0

D. -1

Explanation

$0^{-1} = 1/0 =$???

Assignment

- Refer to lecture slides
  - Assignment: `<var> = <expr>`
  - Simultaneous assignment: `<var1>, <var2>, ..., <varN> = <expr1>, <expr2>, ... <exprN>`
  - Cascading assignment: `<var1> = <var2> = ... = <varN> = <expr>`
  - Augmented assignment: `<var> += <expr>` (similarly for other arithmetic operations)
- Simultaneous assignment and cascading assignment can be used together, e.g. `<var1>, <var2> = <var3>, <var4> = <expr1>, <expr2>`
- Augmented assignment can only be used by itself

Cascading Assignment

- `aa = bb = cc = 1 + 2`
- Equivalent to:

```
temp = 1 + 2  # `1 + 2` is evaluated exactly once
aa = temp
bb = temp
cc = temp
```

- The rightmost hand side expression is evaluated once and saved, and before any assignment to variables
- Order of assignment is from left to right
  - Reverse of that in many other programming languages, e.g. C++, Java, etc.

Simultaneous Assignment

- A special case of unpacking (learn later)
- `aa, bb, cc = 1 + 2, 3, 4`
- Equivalent to:

```
temp0, temp1, temp2 = 1 + 2, 3, 4  # `1 + 2`, `3`, and `4` are each evaluated once
aa = temp0  # aa = 3
bb = temp1  # bb = 3
cc = temp2  # cc = 4
```

- The rightmost hand side expression is evaluated once and saved, and before any assignment to variables
- Order of assignment is from left to right

Cascading Assignment and Simultaneous Assignment

- Cascading assignment and simultaneous assignment can be used together
- Behaviors of both assignments apply
- `aa, bb, cc = xx, yy, zz = 1 + 2, 3, 4`
- Equivalent to:

```
temp0, temp1, temp2 = 1 + 2, 3, 4  # `1 + 2`, `3`, and `4` are each evaluated once
aa = temp0  # aa = 3
bb = temp1  # bb = 3
cc = temp2  # cc = 4
xx = temp0  # xx = 3
yy = temp1  # yy = 3
zz = temp2  # zz = 4
```

- What about...

  `aa, bb, cc = uu = 1 + 2, 3, 4`
  - Revisit this later by yourself once you have learnt collections

Augmented Assignment

- `aa *= 1 + 2`
- Equivalent to: `aa = aa * (1 + 2)`
- The variable must be initialized, as augmented assignment reads the variable first
- The right hand side expression is evaluated once before the assignment
  - Not equivalent to: `aa = aa * 1 + 2`
- Interesting fact: No increment (++aa, aa++) or decrement (--aa, aa--) operators in Python
  - In Python, resort to augmented assignment: `aa += 1`
  - Many other programming languages have them

Exercise: Assignment

Give the values of all variables for each of the following codes. Consider each case separately.

1. `x = 42`
2. `x = x`
3.
```
x = 1023
x = x
```
4. `x = y = 1023`
5. `x, y = 10, 23`
6. (Demo)
```
x, y = 10, 23
y, x = x, y
```
7. (Demo)
```
x = 7
x //= 3
```

Exercise: Assignment (Cont.)

Give the values of all variables for each of the following codes. Consider each case separately.

1. `x = 42`
2. `NameError: name 'x' is not defined`
3. `x = 1023`
4. `x = 1023; y = 1023`
5. `x = 10; y = 23`
6. `x = 23; y = 10`
7. `x = 2`

Exercise: Assignment

Give the values of all variables for each of the following codes. Consider each case separately.

6. `x = 23; y = 10`
   - Idiomatic (or as they say, 'Pythonic') way to swap two (or more) variables
   - Equivalent to:

```
# x, y = 10, 23
temp0, temp1 = 10, 23
x = temp0  # x = 10
y = temp1  # y = 23

# y, x = x, y
temp2, temp3 = x, y
y = temp2  # y = 10
x = temp3  # x = 23
```

iPRS Question 4

Which of the following assignment expressions produce an error?

A. `course = favorite_course = "COMP 1023"`

B. `subject, code = "COMP", "1023"`

C. `comp *= 1023`

D. `course = subject, code = "COMP", "1023"`

iPRS Question 4 - Answer

Which of the following assignment expressions produce an error?

A. `course = favorite_course = "COMP 1023"`

B. `subject, code = "COMP", "1023"`

C. `comp *= 1023`

D. `course = subject, code = "COMP", "1023"`

Explanation

`comp` is uninitialized.

Exercise: Comparison Operators

Refer to lecture slides

- Relational operations
- String comparisons

Evaluate the following expressions involving comparison.

1. `8 < 24`
2. `8 => 24`
3. `1 = 1`
4. `-0.0 != 0.0`
5. `"COMP 1021" == "COMP 1023"`
6. `"COMP 1021" < "COMP 1023"`
7. `"ZZa" < "aZZ"`
8. (Demo) `"ZZa" < "ZZ"`
9. (Demo) `"2" < "10"`
10. (Supplementary) `import math; math.nan == math.nan`

Exercise: Comparison Operators

Evaluate the following expressions involving comparison.

1. `8 < 24`: `True`
2. `8 => 24`: `SyntaxError: cannot assign to literal`
3. `1 = 1`: `SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?`
4. `-0.0 != 0.0`: `False`
5. `"COMP 1021" == "COMP 1023"`: `False`
6. `"COMP 1021" < "COMP 1023"`: `True`
7. `"ZZa" < "aZZ"`: `True`
   - All capital letters are lexicographically smaller than all lowercase letters in ASCII
8. `"ZZa" < "ZZ"`: `False`
   - If a `str` is a prefix of the other `str`, the shorter `str` is lexicographically smaller
9. `"2" < "10"`: `False`
   - Since they are `str`s, we use lexicographic order
   - '2' comes after '1' lexicographically, so the first `str` is lexicographically larger

Exercise: Comparison Operators

Evaluate the following expressions involving comparison.

10. (Supplementary) `import math; math.nan == math.nan`: `False`
    - Surprise!
    - `math.nan` stores a `float` representing NaN (not a number)
    - NaNs are defined to be not equal to itself by IEEE 754
    - Indeed, `math.nan != math.nan` is `True`
    - All other comparison operators (except for `!=`) between two NaNs are also defined to return `False`

Logical Operators

- Refer to lecture slides
  - Logical operations
- Left-associative
- Logical operator precedence
  1. `not`
  2. `and`
  3. `or`
- Logical expressions are more readable when consecutive operations using the same logical operator are parenthesized
  - Even in cases in which it is unnecessary by operator precedence
  - `a or b and c and d or e and f`
  - `a or (b and c and d) or (e and f)`
  - Equivalent, but their readability differs

Exercise: Logical Operators

Evaluate the following expressions involving logical operators.

1. `0 == 9 or 1 == 1`
2. `1 < 2 and 2 < 3`
3. `not (not 1 < 2 or not 2 < 3)`
4. (Demo) `not not not not not 824 == 824`
5. (Demo) `True or True and False`
6. `1 < x or 1 >= x`, where `x` is an `int`

Exercise: Logical Operators

Evaluate the following expressions involving logical operators.

1. `0 == 9 or 1 == 1`: `True`
2. `1 < 2 and 2 < 3`: `True`
3. `not (not 1 < 2 or not 2 < 3)`: `True`
   - `not` has a higher precedence than `and` and `or`
   - This item is equivalent to the previous item; left as an exercise to the reader
4. `not not not not not 824 == 824`: `False`
   - Count the number of `not`s
5. `True or True and False`: `True`
   - `and` has a higher precedence than `or`
   - Equivalent to `True or (True and False)`, but not equivalent to `(True or True) and False`
6. `1 < x or 1 >= x`, where `x` is an `int`: `True`
   - Mathematically, `x` is either smaller than 1 or not smaller than 1

Lab 3

- This is not a hand-in lab: No need to submit anything
- Point-of-sale software
- Two paths: basic, advanced
- Basic: recommended; use a console to accept inputs and print outputs
- Advanced: optional; use a graphical user interface (GUI), like most of your applications
  - Uses Tkinter, a GUI framework that comes with Python

Lab 3

- Basic path: recommended
- Task 1: Get product information from user input
  - How to get user input in a console?
  - How to store user input?
  - What is the original type of user input?
  - What should the correct types be according to the task requirements?
  - How to convert user inputs into the correct types?
- Task 2: Calculate the total cost of one kind of item
  - How many kinds of items are there?
  - Given the unit cost and quantity of an item, how to calculate its total cost?

Lab 3

- Basic path: recommended
- Task 3: Calculate the total cost of both items
  - Given the total cost for each item, how to get the total cost of all items?
- Display the product information and total cost
  - What is the output format required by the task?
  - How to format a string? Note there are multiple ways.
  - What is the best way to format a string in this situation?

Lab 3

- Advanced path: optional
- Same tasks as basic path
- However, each task is on separate function (learn later)
  - Helps to group closely related code together
  - Helps to reuse code
- Run `main_gui.py` after finishing all tasks to open a GUI: `python main_gui.py`
- Interact with the GUI to check your code
- A brief Tkinter tutorial on the lab webpage
  - A GUI framework that comes with Python
  - Not needed to finish the advanced path
  - For learning to create your own GUIs using Python

Use of Numerical Data Types in Python

- `int`, `float`, `decimal.Decimal`
- When and where to use them?
- If your numerical data is integer, use `int`
  - No precision problems
  - Arbitrarily small or large integers can be represented
- Otherwise, use `float` or `decimal.Decimal`, depending on your use case
  - Most applications do not require precision
  - Thus `float` is commonly used
  - Still, apply some caution!

Use of Numerical Data Types in Python

- A few specialized applications require precision
- Banking: If I add 0.1 dollars to my account, then 0.2 dollars, and finally remove 0.3 dollars, I expect 0 dollars left, not `5.551115123125783e-17` dollars.
  - Try running `0.1 + 0.2 - 0.3` in Python REPL
  - An example in later slides
- Finance: ... In January 1982 the index was initialized at 1000 ... Over the weekend of November 25–28, 1983, the error was corrected, raising the value of the index from its Friday closing figure of 524.811 to 1098.892. (Vancouver Stock Exchange §Rounding errors on its Index price)
- Mission-critical systems: On 25 February 1991, a loss of significance in a MIM-104 Patriot missile battery prevented it from intercepting an incoming Scud missile... (Floating-point arithmetic §Incidents)

decimal.Decimal

- Import from the Python builtin package decimal first: `import decimal`, `from decimal import Decimal`, or `from decimal import *`
- Like `float`, it supports all arithmetic and comparison operators
- No precision loss up to a user-specified precision (number of accurate digits)
  - By default this is 28
  - Can be set using `decimal.getcontext().prec = desired_precision`
- Supports conversion from `int`, `float`, `str`, ...
  - Converting from a `float` uses its exact value, including any precision loss
    - `Decimal(3.14)` is `Decimal('3.14000000000000012434497875801753252744674682617187 5')`, because 3.14 cannot be represented exactly using `float`
  - Convert from a `str` instead to avoid this
    - `Decimal("3.14")` is `Decimal('3.14')`
- Explore by yourself!

decimal.Decimal

- Revisit the banking example
- Bad version using `float`:

```
balance = 0.0
while True:
    print("You have", balance, "dollars in your bank account")
    change = float(input("Deposit (or withdraw) money: "))
    balance += change
```

- Better version using `decimal.Decimal`:

```
import decimal
balance = decimal.Decimal("0")
while True:
    print("You have", balance, "dollars in your bank account")
    change = decimal.Decimal(input("Deposit (or withdraw) money: "))
    balance += change
```

- For both versions: deposit 0.1 dollars, and then deposit 0.2 dollars
- Exit the program by pressing `Ctrl+C`

Other Things

- Topics covered in lectures but not this tutorial
- Refer to lecture slides
  - Introduction
  - Operator precedence table
  - Division by zero
  - Chain operators: also see appendix
  - Membership operations
  - Floating point errors
  - Common pitfalls

That's all!

Any question?

Appendix

- These materials are optional
- Feel free to read them if you are interested

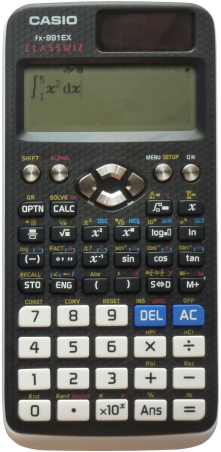
A Brief Etymology of ‘Computer’

- ‘Computer’ used to refer to a person who carries out calculations
- It was not until the mid-20th century that ‘computer’ acquired its current definition
- Nowadays computers (its modern definition) can perform billions of calculations every second!
- You do not need to be a ‘human computer’, but you should definitely know how to instruct computers to perform arithmetic operations

NACA High Speed Flight Station ‘Computer Room’

A Physical Calculator

- A calculator may include:
  - Basic arithmetic: addition ($+$), subtraction ($-$), multiplication ($\times$), division ($\div$), exponentiation, ...
  - Elementary functions: sine, cosine, tangent, their inverses, logarithm, ...
  - Calculus: differentiation, integration, ...
  - Statistics: max, min, mean, median, variance, linear regression, ...
  - Graphics: graph of a function, ...
  - Memory and programs
  - ...and more!
- Python (with the appropriate packages) can also perform them as well!
  - With great power comes with great complexity...
  - For simplicity, let's only consider basic arithmetic first!



Return Type of Arithmetic Operators

- Rules of thumb to help memorize return type (lower number is more important)
  1. It should reasonably represent the exact result
  2. Return `int` if both numbers are `int`, otherwise `float`
- Examples
  - `3 / 2`: `1.5` (`float`); division in general does not produce an integer value, so return `float`
  - `3 // 2`: `1` (`int`); floor division in general produce an integer value, so return `int`
  - `3.0 // 2`: `1.0` (`float`); one operand is `float`, so return `float` even if the result is an integer value
  - `1 ** -1`: `1.0` (`float`); taking the negative power requires division

Order of Operations

- Refer to lecture slides
  - Precedence of arithmetic operators
- Python follows the same convention as that we humans normally use:
- Mnemonics for order of operations
  - PEMDAS: Parentheses, Exponents, Multiplication/Division, Addition/Subtraction
    - Please Excuse My Dear Aunt Sally
  - BODMAS: Brackets, Of, Division/Multiplication, Addition/Subtraction
    - "Of" meaning fraction multiplication
  - 先乘除後加減: "first multiply and divide, then add and subtract"

Short Circuiting of Logical Operators

- To disable short circuiting...
  1. Any non-`bool` expressions should be converted to `bool` first using `bool(<expr>)`
  2. Replace `and` with `&`
  3. Replace `or` with `|`
  4. Add parentheses to ensure the order of operations is the same as before, as `&` and `|` have higher operator precedences
- Examples
  - `x or y` becomes:
    ```
    x | y
    ```
  - `x + 1 == y and z + 1 == w` becomes:
    ```
    (x + 1 == y) & (z + 1 == w)
    ```
  - For exposition only: `a_list and a_tuple` becomes:
    ```
    bool(a_list) & bool(a_tuple)
    ```

Chaining Comparison Operators

- Refer to lecture slides
  - Chain operators
- It allows you to write intuitive comparison statements such as `2 < x < y < 4`
  - `True` if `x` is between 2 and `y` (excluding 2 and `y`) and `y` is between `x` and 4 (excluding `x` and 4)
- It also allows you to write nonsensical comparison statements such as `-2 < x < y < -4`
  - Don't do this in practice!
- Many other programming languages do not allow this, e.g. C++, Java, ...
- An arbitrary number of arbitrary comparison operators can be chained (but usually not a good idea)
  - `2 < x < 4`
  - `2 < x < y < 4`
  - `2 == x < y == 4`
    - No good reason to write code like this in practice apart from exams...
- For the general form: see appendix

Chaining Comparison Operators

- General form
  - `<expr1> <op1> <expr2> <op2> ...<opN-1> <exprN> <opN> <exprN+1>`
- Interpretation
  - Each `<opK>` corresponds to a comparison
    - `<expr1> <op1> <expr2>`
    - `<expr2> <op2> <expr3>`
    - ...
    - `<exprN> <opN> <exprN+1>`
  - All of them must be `True` for the entire logical expression to be `True`; otherwise it is `False`
    - You can think that each of the above comparisons are parenthesized, and then joined together using `and`
  - Each `<exprK>` is evaluated exactly once
    - Ignore this for now if you do not understand; it won't matter yet

Chaining Comparison Operators

- The conversion is imperfect: Each <exprK> should be evaluated exactly once, but this is not the case after the conversion; see appendix
- `2 < x < 4`: `(2 < x) and (x < 4)`
- `2 < x < y < 4`: `(2 < x) and (x < y) and (y < 4)`
- `2 == x < y == 4`: `(2 == x) and (x < y) and (y == 4)`
- `2 < x < y < 4 > 6 != 7 == 9 >= -1 != 2 > 2`:

```
(
      (2 < x)   and (x < y)   and (y < 4)
  and (4 > 6)   and (6 != 7)  and (7 == 9)
  and (9 >= -1) and (-1 != 2) and (2 > 2)
)
```

Exercise: Chaining Comparison Operators

Evaluate the following expressions involving chained comparison operators.

1. `2 < 3 < 4`
2. `"COMP 1021" < "COMP 1023" < "COMP 2011" < "COMP 2012"`
3. `2 < x > 2`, where `x` is an `int`

Exercise: Chaining Comparison Operators

Evaluate the following expressions involving chained comparison operators.

1. `2 < 3 < 4`: `True`
2. `"COMP 1021" < "COMP 1023" < "COMP 2011" < "COMP 2012"`: `True`
3. `2 < x > 2`, where `x` is an `int`: `False`
   - Mathematically, `x` cannot be simultaneously smaller than 2 and larger than 2

Meaning of "Evaluated Once"

- In assignment and chaining comparison operators, "evaluated once" is mentioned
- What does it mean? Why does it matter?
- Consider the following program:

```
if 0 <= float(input("Enter a number: ")) <= 10:
  print("Your number is in between 0 and 10!")
else:
  print("Your number is not in between 0 and 10.")
```

- `float(input("Enter a number:" ))` is "evaluated once"
- If you run the above program, you are prompted for a number once
- Consider the following "equivalent" program:

```
if 0 <= float(input("Enter a number: ")) and float(input("Enter a number: ")) <= 10:
  print("Your number is in between 0 and 10!")
else:
  print("Your number is not in between 0 and 10.")
```

- If you run the above program instead, you are prompted for a number twice

Meaning of "Evaluated Once"

- The example relates to chaining comparison operators, but the concept applies to assignment too
- "Evaluated once" matters when the expression has side effects
  - The expression `1 + 2` has no side effects; running it once or multiple times does not affect the program behavior to the user (apart from performance)
  - The expression `input("Enter a number: ")` has side effects; running it once or multiple times prompts the user for a number of a different number of times
- Most of the expressions we have learnt so far do not have side effects
- This is why this is supplementary content, because it likely does not matter for now
- Later, you will learn more expressions with side effects

Value Equality and Object Equality

- Computer memory stores many objects; each object has a memory address
  - A city has many houses; each house has an address
- If two variables refer to the same memory address, they must refer to the same object
- The identity operator `is` (`is not`) tells us if this is the case; it checks for object equality
- The equals operator == checks for value equality

Value Equality and Object Equality

- Semantically, they are different (their meanings are different)
- An object should equal the object itself in value
  - Object equality implies value equality
  - Side note: Python is naughty... you can change the behavior of `==` to always return `False` or do other weird things
- Different objects can have the same value
  - Value equality may not imply object equality
  - Imagine two `int` objects at different memory address, but they have the same value of 1023

Special Values and Cached Values

- Special values with unique object or memory address
  - Python guarantees only one object with this value can ever exists
  - Thus in this special case, value equality does imply object equality
  - Examples: None (`NoneType`), Ellipsis or `...` (`EllipsisType`), ...
- Cached values
  - Whenever appropriate, Python avoids creating new objects, and reuse existing ones with the same value as the object to be created to save memory
  - An implementation detail of Python; not guaranteed
  - Don't rely on it for program correctness
    - Python may print a `SyntaxWarning` when you rely on it
    - Try running `0 is 0` in Python REPL
  - Examples: small `ints`, short `strs`, ...