Flow of Control

Python Programming

Byeongjoon Noh

powernoh@sch.ac.kr

SCH SOON CHUN HYANG UNIVERSITY

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

Contents

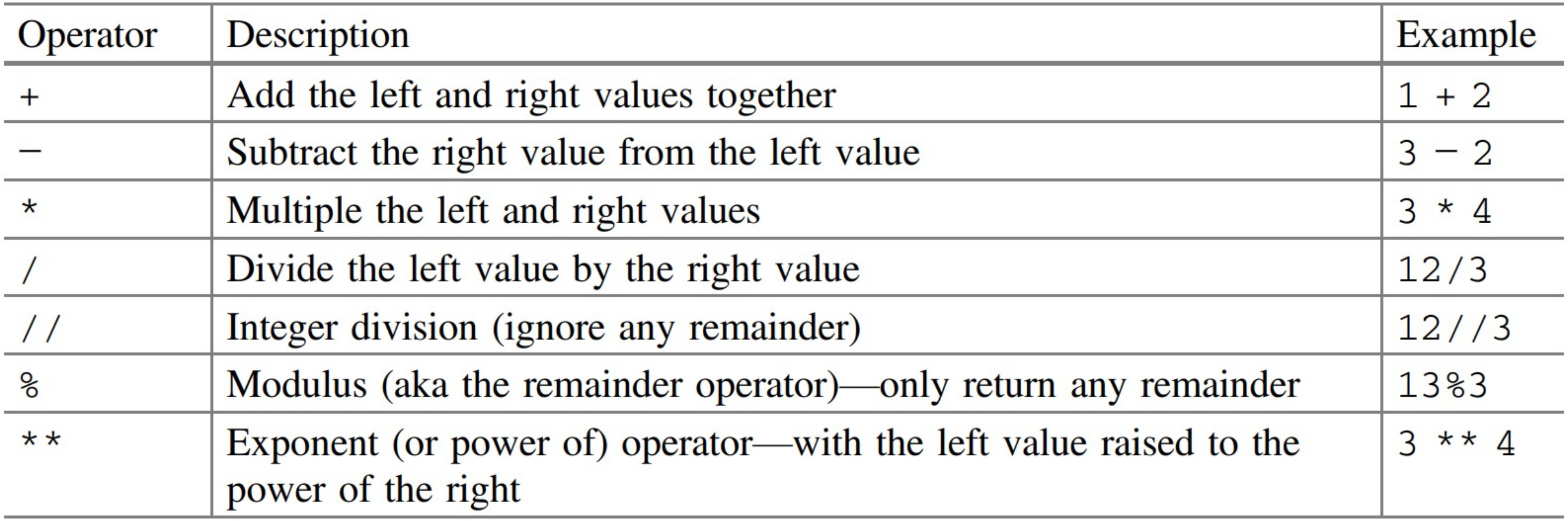
1. Basic operators
2. Flow of control using “if” statements
3. Iteration and looping
4. Error and exception handling

Textbook: Chapter 5.7, 5.8, 5.9, Chapter 6, Chapter 7, Chapter 24

1. Basic operators

Types of arithmetic operators

- Arithmetic operators
  - used to perform some form of mathematical operation
    - e.g., addition, subtraction, multiplication and division etc.
  - in Python, they are represented by one or two characters as follows:



Operator	Description	Example
+	Add the left and right values together	`1 + 2`
−	Subtract the right value from the left value	`3 − 2`
*	Multiple the left and right values	`3 * 4`
/	Divide the left value by the right value	`12/3`
//	Integer division (ignore any remainder)	`12//3`
%	Modulus (aka the remainder operator)—only return any remainder	`13%3`
**	Exponent (or power of) operator—with the left value raised to the power of the right	`3 ** 4`

Integer operations

- Two integers can be added together using +, - and *
  - Operations such as +, - and * between integers always produce integer results

```
home = 10
away = 15
print(home + away)
print(type(home + away))
print(10 * 4)
print(type(10*4))
goals_for = 10
goals_against = 7
print(goals_for - goals_against)
print(type(goals_for - goals_against))
```

```
25
<class 'int'>
40
<class 'int'>
3
<class 'int'>
```

Integer operations

- Division operator (/)
  - 100 / 20 ➔ reasonably expect to produce might be 5; but actually 5.0

```
print(100 / 20)
print(type(100 / 20))
```

```
5.0
<class 'float'>
```

  - Because the computer cannot the result of division operation in advance; so designate floating point number by default

```
res1 = 3/2
print(res1)
print(type(res1))
```

```
1.5
<class 'float'>
```

Integer operations

- Integer division operator (//)
  - ignoring the fractional part then there is an alternative version of the divide operator

```
res1 = 3//2
print(res1)
print(type(res1))
```

```
1
<class 'int'>
```

- Modulus operator (%)
  - returns the remainder of a division operation

```
print('Modulus division 4 % 2:', 4 % 2)
print('Modulus division 3 % 2:', 3 % 2)
```

```
Modulus division 4 % 2: 0
Modulus division 3 % 2: 1
```

Integer operations

- Power operator (**)
  - to raise an integer by a given power
    - 5**3 means 5^3

```
a = 5
b = 3
print(a ** b)
```

```
125
```

  - in fact, these two operands have also floating point numbers

```
a = 5
b = 0.5
print(a ** b)
```

```
2.23606797749979
```

Floating point number operations

- Multiple, subtract, add and divide operations available for floating point numbers
  - All these operators produce new floating point numbers

```
print(2.3 + 1.5)
print(1.5 / 2.3)
print(1.5 * 2.3)
print(2.3 - 1.5)
print(1.5 - 2.3)
```

```
3.8
0.6521739130434783
3.4499999999999997
0.7999999999999998
-0.7999999999999998
```

Floating point number operations

- Any operation involving both integers and floating point numbers ➔ will produce a floating point number
  - if one of the sides of an operation such as add, subtract, divide or multiple is a floating point number, then the result will be a floating point number

```
i = 3 * 0.1
print(i)
```

```
0.30000000000000004
```

  - Which may or may not have been what you expected; 0.3
    - floating point number being presented as an approximation within a computer system
    - solution) use `decimal` module

Floating point number operations

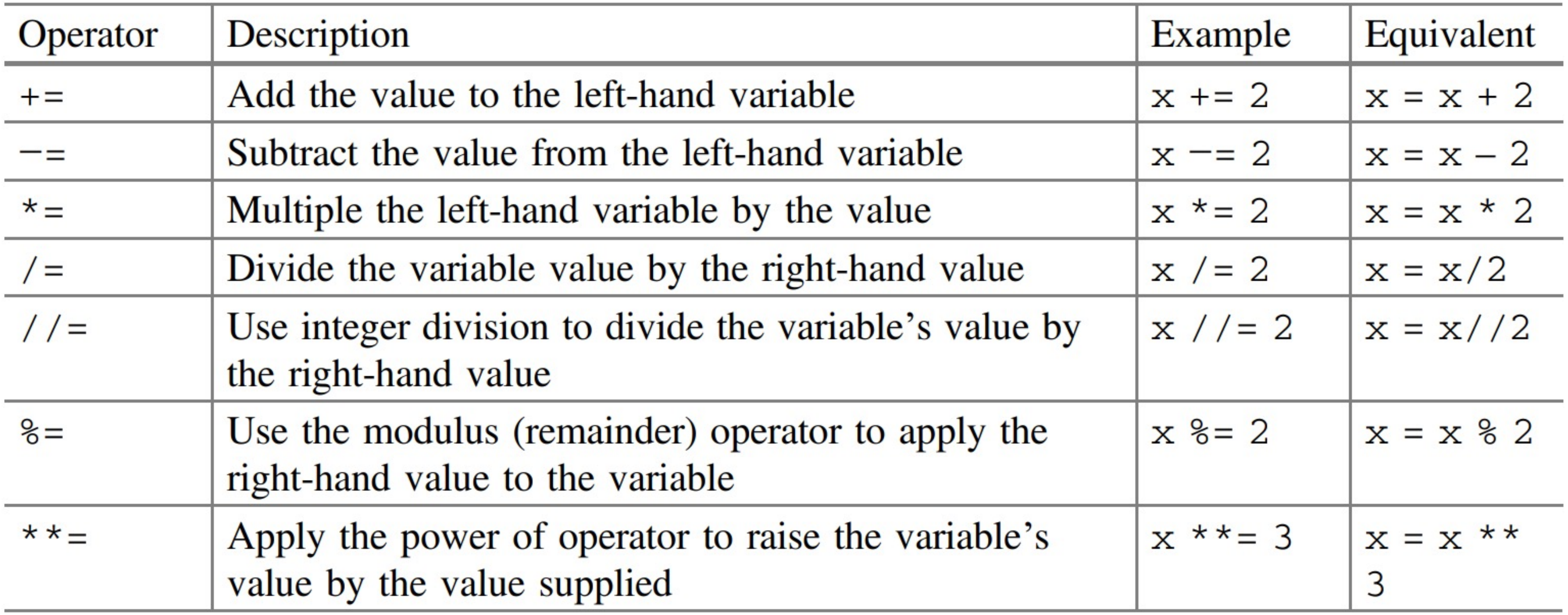
- Ceiling and flooring operation
  - to adjust the real numbers to the nearest integer up or down
  - need to import 'math' module
  - ceiling: math.ceil()
    - find the smallest integer greater than or equal to the number
  - flooring: math.floor()
    - find the largest integer less than or equal to the number

```
import math

print(math.ceil(2.3))  # Outputs: 3
print(math.ceil(-2.3))  # Outputs: -2
print(math.floor(2.3))  # Outputs: 2
print(math.floor(-2.3))  # Outputs: -3
```

Assignment operators

- To assign a value to a variable
  - the available compound operators in Python

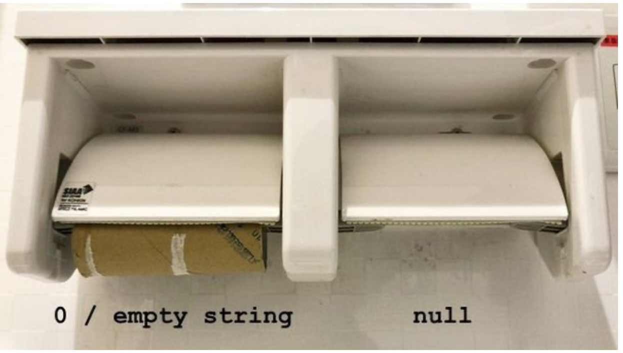


Operator	Description	Example	Equivalent
+=	Add the value to the left-hand variable	x += 2	x = x + 2
−=	Subtract the value from the left-hand variable	x −= 2	x = x − 2
*=	Multiple the left-hand variable by the value	x *= 2	x = x * 2
/=	Divide the variable value by the right-hand value	x /= 2	x = x/2
//=	Use integer division to divide the variable's value by the right-hand value	x //= 2	x = x//2
%=	Use the modulus (remainder) operator to apply the right-hand value to the variable	x %= 2	x = x % 2
**=	Apply the power of operator to raise the variable's value by the value supplied	x **= 3	x = x ** 3

```
x = 0
x += 1 # has the same behavior as x = x + 1
```

None value

- A special type in Python; None
  - <NoneType> with a single value
  - to represent null values or *nothingness*
    - Different with False, or empty string or 0
  - can be used when you need to create a variable, but don't have an initial value for it



```
winner = None
print(type(winner))
```

```
<class 'NoneType'>
```

- Test for the presence of None using 'is' and 'is not'

```
print(winner is None)
print(winner is not None)
```

```
True
False
```

Quiz

- What is the output of the following code?

```python
str1 = "abc"
str2 = str1
str1 += "d"
print(str1 == str2)
```

  - a) True
  - b) False
  - c) Error
  - d) None

Quiz

- What is the output of the following code?

```
print(3%-2)
```

  - a) 1
  - b) 0
  - c) -1
  - d) Error

Quiz

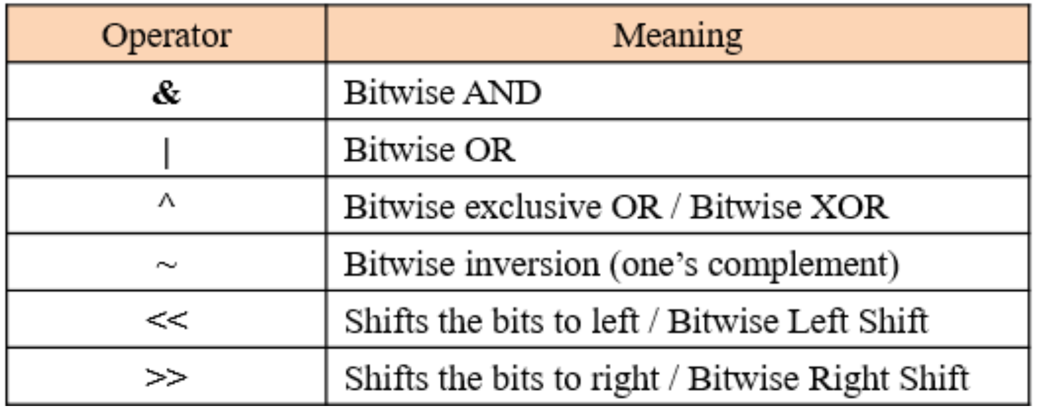
- What is the output of the following code?

```
print(3*2**3)
```

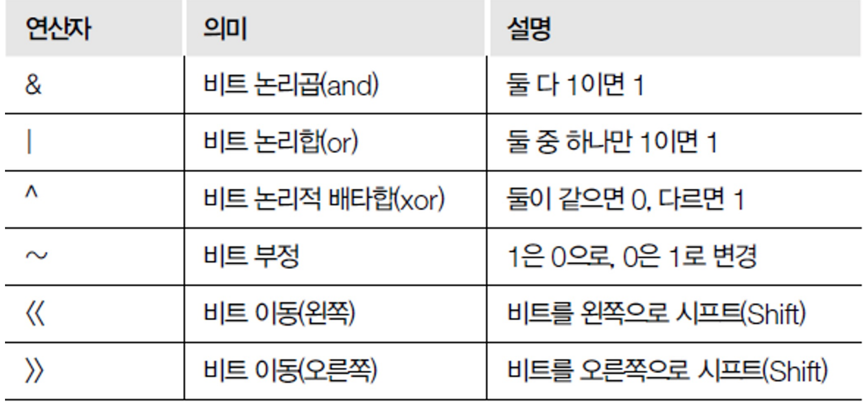
  - a) 48
  - b) 24
  - c) 64
  - d) 18

Note: Bitwise operators

- Used to perform operations on binary numbers at the bit level
  - These operators treat their operands as sequences of 64 bits, and operate on them bit by bit



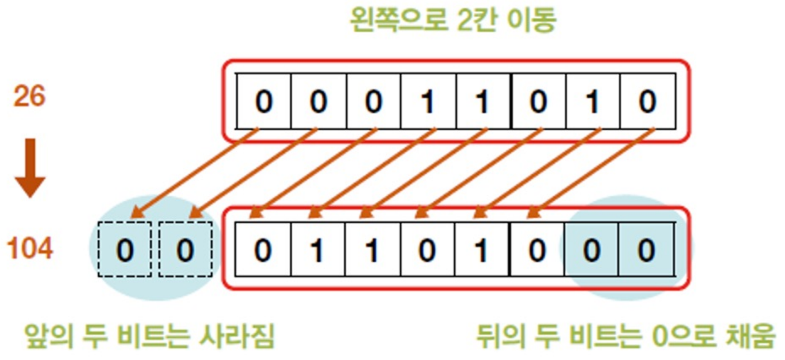
Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR / Bitwise XOR
~	Bitwise inversion (one’s complement)
<<	Shifts the bits to left / Bitwise Left Shift
>>	Shifts the bits to right / Bitwise Right Shift



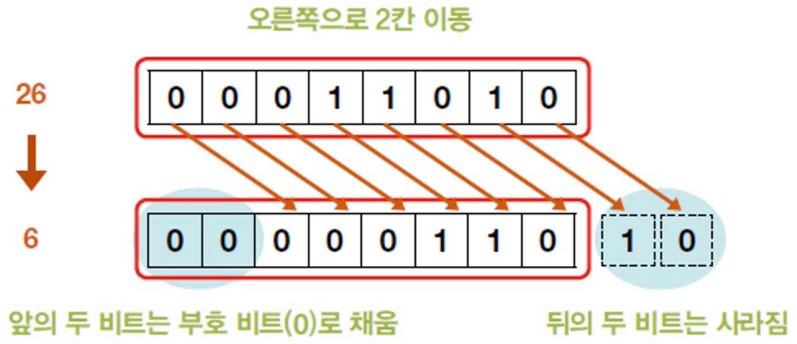
연산자	의미	설명
&	비트 논리곱(and)	둘 다 1이면 1
|	비트 논리합(or)	둘 중 하나만 1이면 1
^	비트 논리적 배타합(xor)	둘이 같으면 0, 다르면 1
~	비트 부정	1은 0으로, 0은 1로 변경
《	비트 이동(왼쪽)	비트를 왼쪽으로 시프트(Shift)
》	비트 이동(오른쪽)	비트를 오른쪽으로 시프트(Shift)

Note: Bitwise operators

- << operator (left shift operator)
  - Shifts the bits to the left by a specified number of places (fills in with 0s on the right)
  - effectively multiplies by 2ˆ(n) with n times shift to the left



- >> operator
  - Shifts the bits to the right by a specified number of places (fills in with the sign bit on the left in case of signed numbers)
  - effectively multiplies by 2ˆ(-n) with n times shift to the right



Note: Bitwise operators

- Example of bitwise operators

```
a = 50      # 110010
b = 25      # 011001
print(a & b)
print(a | b)
print(a ^ b)
print(~a)
print(~a+1) # convert to 2’s complement
print(a << 2)
print(a >> 2)
```

```
16
59
43
-51
-50
200
12
```

In class practice

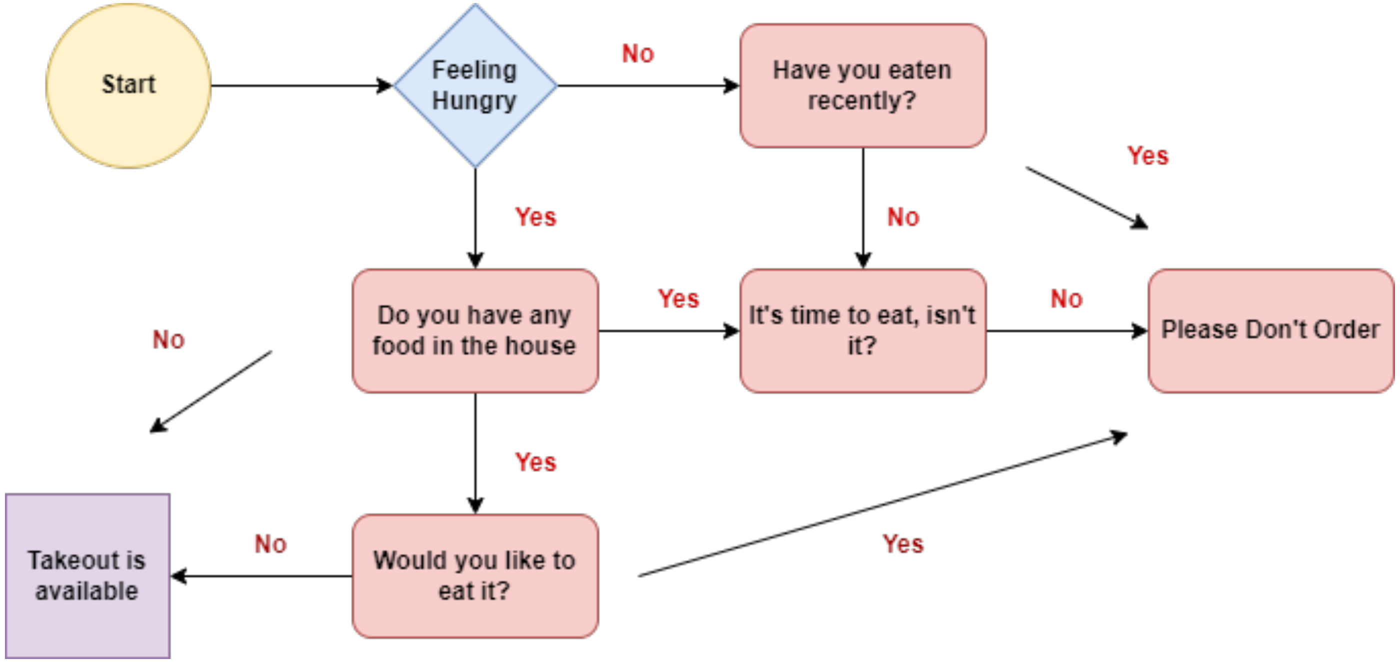
- P03-01 사용자로부터 kilometer의 값을 입력받아서 mile로 변환하는 프로그램을 작성해보세요.
  - requirements
    - `input()` function을 사용하여 사용자로부터 값을 입력받을 것
    - mile = 0.6214 * kilometers
  - input: kilometer value
  - output: mile value

```
Enter the kilometer: 1758
1758 kilometer is 1092.4212 miles
```

2. Flow of control using “if” statements

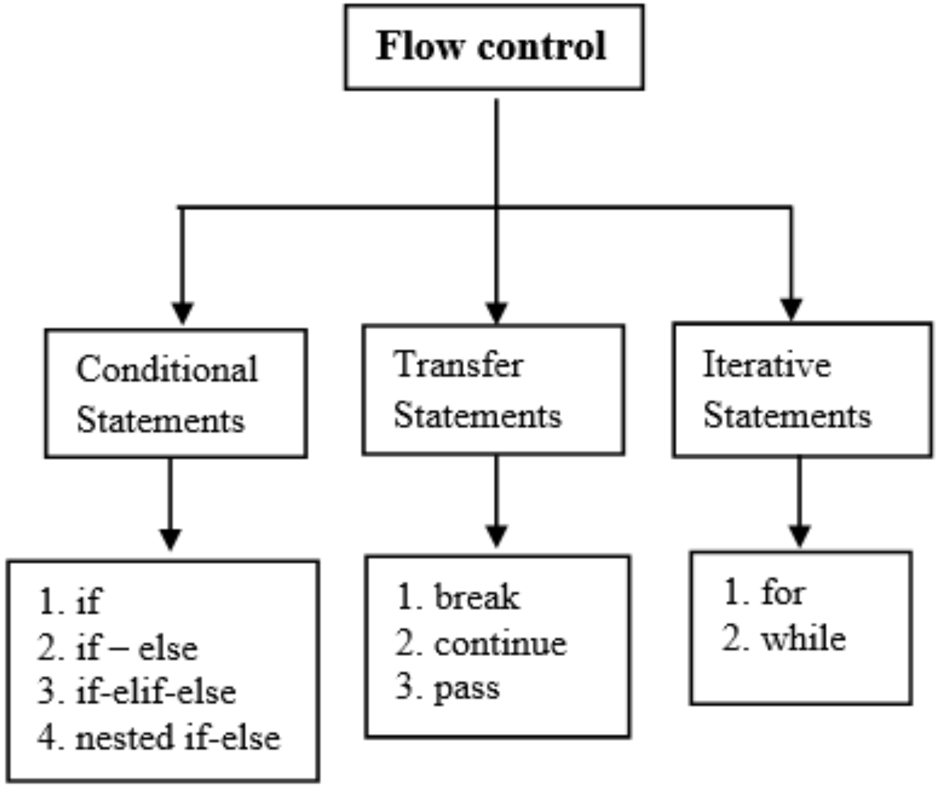
Flow control

- “Flow control” determine how a program will respond to different condition and decide which path of execution to follow
  - refers to the order in which individual statements, instructions, or function calls
  - a fundamental concept in programming that directs the order of operations based on logical rules and conditions



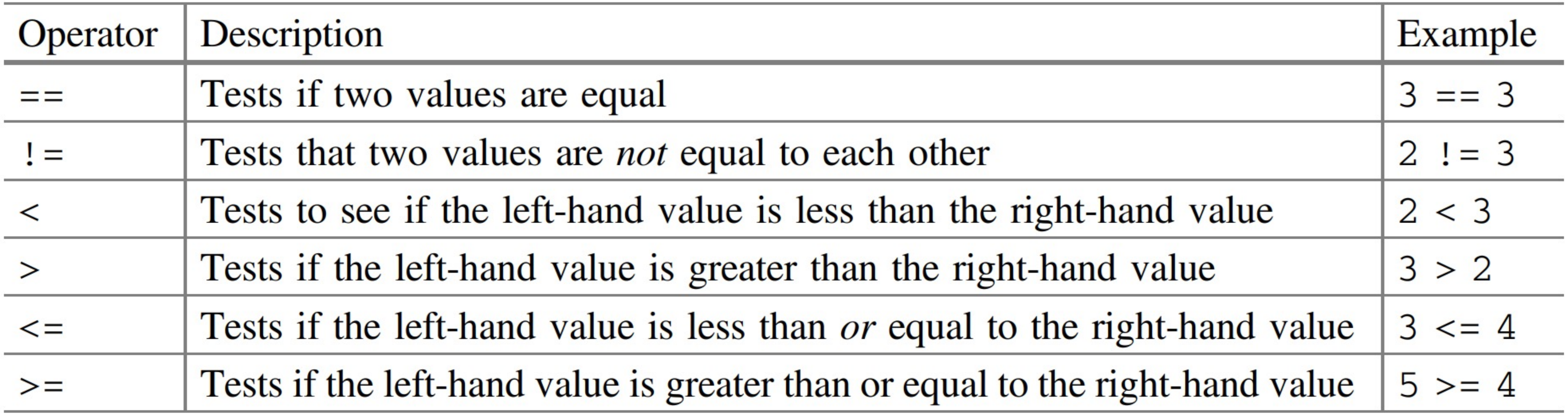
Flow control

- There are mainly three statements to control flow
  - Conditional statements
  - Transfer statements
  - Iterative statements



Comparison operators

- These are operators that return Boolean values; `True` or `False`
    - key to the conditional elements of flow of control statements such as “if”



Operator	Description	Example
`==`	Tests if two values are equal	`3 == 3`
`!=`	Tests that two values are *not* equal to each other	`2 != 3`
`<`	Tests to see if the left-hand value is less than the right-hand value	`2 < 3`
`>`	Tests if the left-hand value is greater than the right-hand value	`3 > 2`
`<=`	Tests if the left-hand value is less than *or* equal to the right-hand value	`3 <= 4`
`>=`	Tests if the left-hand value is greater than or equal to the right-hand value	`5 >= 4`

- used in everyday life all the time
    - do I have enough money to buy lunch, or is this shoe in my size, etc.

Comparison operators

```
a, b = 100, 200
print(a == b)
print(a != b)
print(a > b)
print(a <= b)
```

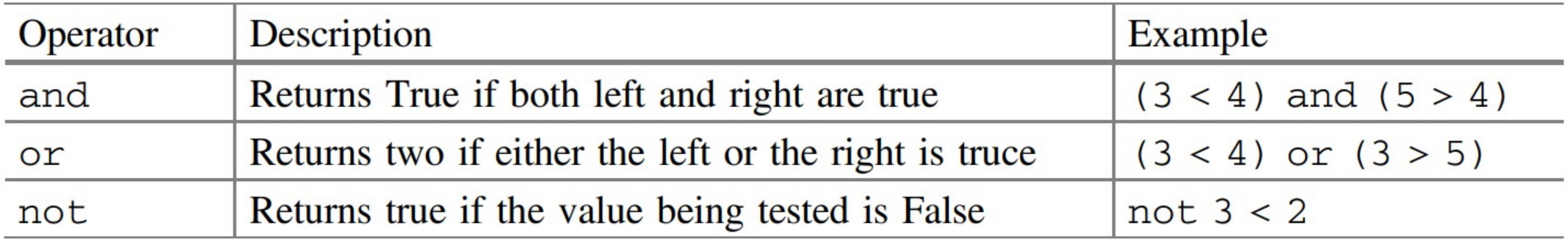
```
False
True
False
True
```

```
name1 = "John is nice."
name2 = "john is nice."
print(name1 == name2)
name2 = "John is nice."
print(name1 == name2)
```

```
False
True
```

Logical operators

- Used to combined Boolean expressions together
  - typically, they are used with comparison operators to create more complex conditions



Operator	Description	Example
and	Returns True if both left and right are true	(3 < 4) and (5 > 4)
or	Returns two if either the left or the right is truce	(3 < 4) or (3 > 5)
not	Returns true if the value being tested is False	not 3 < 2

  - ex) how to express ‘100 < a < 200’

```
(a > 100) and (a < 200)
a > 100 and a < 200
```

  - ex) how to express ‘a < b < c’

```
(a < b) and (b < c)
```

Comparison and logical operators

```
a = 99
print((a > 100) and (a < 200))
print((a > 100) or (a < 200))
print(not(a == 100))
print(not(a != 100))
```

```
False
True
True
False
```

Quiz

- What is the output of the following code?

```
a = 'Hello'
b = 'Hello'
print(f"a is b: {a is b}")
print(f"a == b: {a == b}")
```

  - a) a is b: True

    a == b: False
  - b) a is b: False

    a == b: True
  - c) a is b: True

    a == b: True
  - d) Error

The if statement

- A form of conditional programming;
    - something you probably do every day in the real world

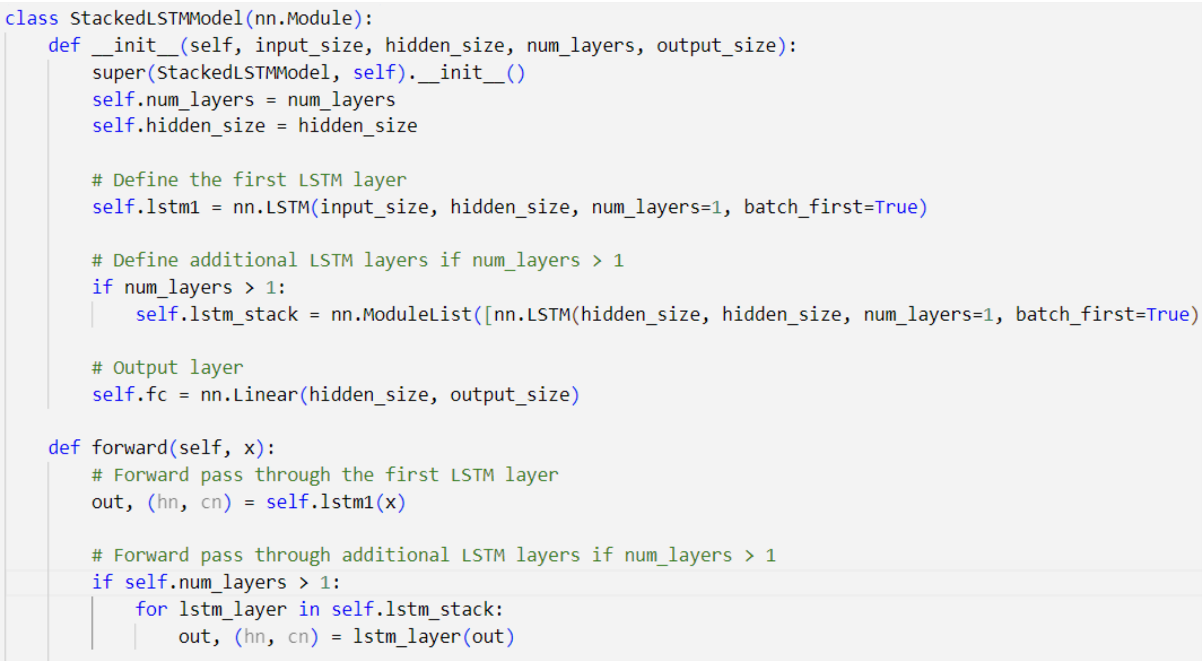
- Syntax (most basic form)

```
if <condition-evaluating-to-boolean>:
    statement
```

    - if the condition is `True` then we will execute the indented statement
    - * Indentation to separate a block for if statement

Note: Indentation in Python

- Importance of Indentation
  - Python uses indentation to define blocks
    - unlike many other programming languages uses braces '{ }' to define a block of code
  - All the code within an if statement, loop, function definition, or any other block must be consistently indented to be considered part of the same block



```
class StackedLSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(StackedLSTMModel, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size

        # Define the first LSTM layer
        self.lstm1 = nn.LSTM(input_size, hidden_size, num_layers=1, batch_first=True)

        # Define additional LSTM layers if num_layers > 1
        if num_layers > 1:
            self.lstm_stack = nn.ModuleList([nn.LSTM(hidden_size, hidden_size, num_layers=1, batch_first=True)

        # Output layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Forward pass through the first LSTM layer
        out, (hn, cn) = self.lstm1(x)

        # Forward pass through additional LSTM layers if num_layers > 1
        if self.num_layers > 1:
            for lstm_layer in self.lstm_stack:
                out, (hn, cn) = lstm_layer(out)
```

- General indentation in Python
  - 4 spaces or 1 tab
    - depending on Python-supported IDE

Working with an “if” statement

- if less than zero a message noting this will be printed to the user

```
num = int(input('Enter a number: '))
if num < 0:
    print(num, 'is negative')
```

```
Enter a number: -10
-10 is negative
```

  - to execute multiple statements when our condition is true
    - we can indent several lines

```
num = int(input('Enter another number: '))
if num > 0:
    print(num, 'is positive’)
    print(num, 'squared is ', num * num)
print('Bye')
```

```
Enter another number: 15
15 is positive
15 squared is  225
Bye
```

“else” in an “if” statement

- An optional element that can be run if the conditional part of the `if` statement returns `False`

```
num = int(input('Enter yet another number: '))
if num < 0:
    print('Its negative')
else:
    print('Its not negative')
```

```
Enter yet another number: 20
Its not negative
```

  - Guaranteed that at least one (and at most one) of the `print()` function will execute

The use of “elif”

- `else-if` scenario
    - In some cases there may be several conditions you want to test, with each condition being tested in the previous one failed
    - by the `elif` element of an `if` statement
        - ➔ follows the `if` part and comes before any (optional) `else` part
    - syntax

```
elif <condition-evaluating-to-boolean>:
    statement
```

The use of "elif"

```python
savings = float(input("Enter how much you have in savings: "))
if savings == 0:
    print("Sorry no savings")
elif savings < 500:
    print('Well done')
elif savings < 1000:
    print('Thats a tidy sum')
elif savings < 10000:
    print('Welcome Sir!')
else:
    print('Thank you')
```

```
Enter how much you have in savings: 500
Thats a tidy sum
```

- the first `if` condition failed (as savings is not equal to 0),
- the next `elif` also must have returned `False` as savings were greater than 500,
- it was second `elif` statement that returned `True` and thus the associated `print()`

Nested if statement

- It is possible to *nest* one `if` statement inside another
  - *nesting*: indicates that one `if` statement is located within part of the another `if` statement and can be used to refine the conditional behaviour of the program

```
snowing = True
temp = -1
if temp < 0:
    print('It is freezing')
    if snowing:
        print('Put on boots')
        print('Time for Hot Chocolate')
print('Bye')
```

```
It is freezing
Put on boots
Time for Hot Chocolate
Bye
```

Short hand form of if statement

- Quite common to want to assign a specific value to a variable dependent on some conditions

- Syntax

```
<result1> if <condition-is-met> else <result2>
```

  - example

```
age = 15
status = None
if (age > 12) and age < 20:
    status = 'teenager'
else:
    status = 'not teenager'
print(status)
```

```
age = 15
status = 'teenager' if age > 12 and age < 20 else 'not teenager'
print(status)
```

Quiz

- What is the output of the following code?

```
x, y = 15, 10
result = x if x < y else y
print(result)
```

  - a) 15
  - b) 10
  - c) False
  - d) Error

In class practice

- P03-02 사용자로부터 정수 1개를 입력받고, 해당 수가 양수 인지 음수인지 0인지 판단하는 프로그램을 작성해보세요.
    - input: 1개의 정수
    - output: 양수, 음수 또는 0

In class practice

- P03-03 사용자로부터 정수 1개를 입력받고 해당 수가 짝수인지 음수인지 판단하여 출력하는 프로그램을 작성해보세요.
    - input: 1개의 정수
    - output: 짝수 또는 홀수
    - hint

```
(num % 2) == 0
```

In class practice

- P03-04 사용자로부터 점수를 입력받고 해당 점수가 pass인지 fail인지 판단하여 출력하는 프로그램을 작성해보세요.
    - requirements
        - if score is greater than 60, print out ‘pass’ message
        - USE short hand form of if statement
    - input: 점수
    - output: ‘pass’ or ‘fail’

```
<result1> if <condition-is-met> else <result2>
```

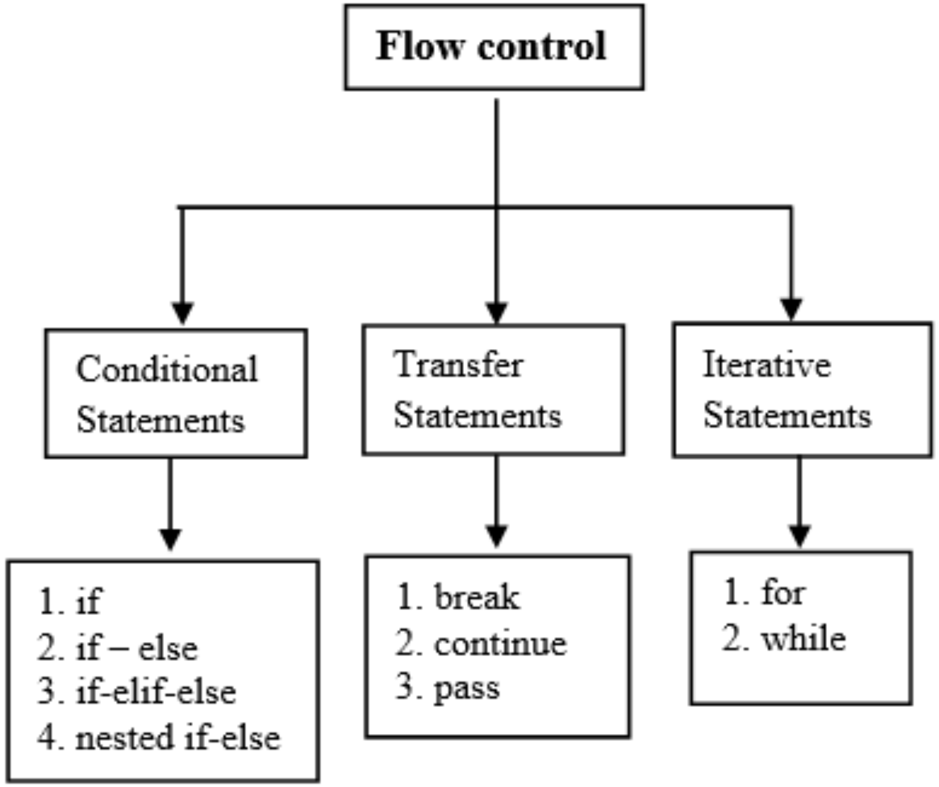
In class practice

- P03-05 사용자로부터 점수를 입력받고, 학점을 A, B, C, D and F로 구분하여 출력하는 프로그램을 작성해보세요.
    - input: a number (grade)
    - output: a letter (grade category)
    - requirements
        - 90 <= A <= 100
        - 80 <= B < 90
        - 70 <= C < 80
        - 60 <= D < 70
        - F < 60

3. Iteration and looping

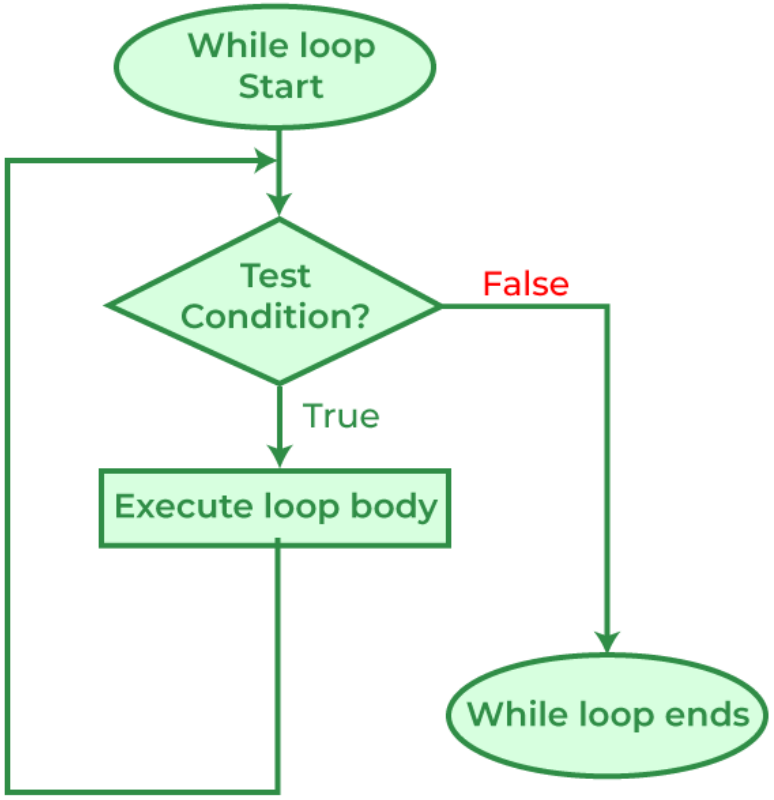
Introduction

- To control the repeated execution of selected statements
  - `while` loop and `for` loop available in Python



While loop

- The while loop exists in almost all programming languages and is used to iterative (or repeat) one or more code statements as long as the test condition (expression) is `True`



While loop

- General syntax

```
while <test-condition-is-true>:
    statement or statements
```

  - test condition/expression is `True` then the statement or block of statements will be executed

- Test is performed before each iteration;

  - if the condition fails the first time around the loop the statement or block of statement may never be executed at all

```
count = 0
print('Starting')
while count < 10:
    print(count, ' ', end='')
    count += 1
print() # not part of the while loop
print('Done')
```

```
Starting
0  1  2  3  4  5  6  7  8  9
Done
```

Quiz

- What is the output of the following code?

```
j = 1
while j <= 2:
    print(j, end = ' ')
    j +=1
```

  - a) 1 2
  - b) 1 2 3
  - c) 1
  - d) None

Note: end='' in print() function

```
print(count, ' ', end='')
```

- print() function ends with a newline character (₩n), which means that after the text in printed, the cursor will move to the next line
- The end='' argument (option) specify; not to end with a newline, but with an empty string instead

For loop

- A far more concise way to make loop
  - typically clearer to another programmer that the loop must iterate for a specific number of iterations
- General syntax

```
for <variable-name> in range(...):
    statement or statements
```

For loop

```
print('Print out values in a range')
for i in range(0, 10):
    print(i, ' ', end='')
print()
print('Done')
```

```
Print out values in a range
0  1  2  3  4  5  6  7  8  9
Done
```

- `range(start, end, step)`
  - range(0, 10); ‘i’ would take values 0, 1, 2, ⋯ up to 9
  - range(0, 10, 2); take 0 to 9 step by 2

```
for i in range(0, 10, 2):
    print(i, ' ', end='')
```

```
0 2 4 6 8
```

For loop

- range(start, end, step)
  - start is also optional

```python
for i in range(4):
    print(i, ' ', end='')
```

```
0 1 2 3
```

For loop

- One interesting variation on for loop is the use of a wild card (‘_’) instead of a lopping variable;
  - this can be useful if you are only interested in looping a certain number of tiems and not in the value of the loop counter itself

```
# Now use an 'anonymous' loop variable
for _ in range(0, 10):
    print('.', end='')
print()
```

  - in this case we are not interested in the values generated by the range; only in looping 10 times thus there is no benefit in recording the loop variable

Quiz

- What is the output of the following code?

```
for i in range(4):
    print(0.1 + i * 0.25, end='')
```

  - a) 0.100.350.60.85
  - b) 0.10.350.60.851.1
  - c) 0.10 0.35 0.6 0.85
  - d) 0.1 0.35 0.6 0.85 1.1

Quiz

- What is the output of the following code?

```python
for i in range(20, 10, -3):
    print(i, end=' ')
```

  - a) 19 16 13 10
  - b) 10 13 16 19
  - c) 11 14 17 20
  - d) 20 17 14 11

In class practice

- P03-06-A: Asterisks (*)을 사용하여 사용자로부터 입력받은 크기의 정사각형을 출력하는 프로그램을 작성해보세요.
  - input: N (사용자로부터 입력받은 정사각형의 한 변의 길이)
  - output: *로 구성된 N*N 크기의 정사각형

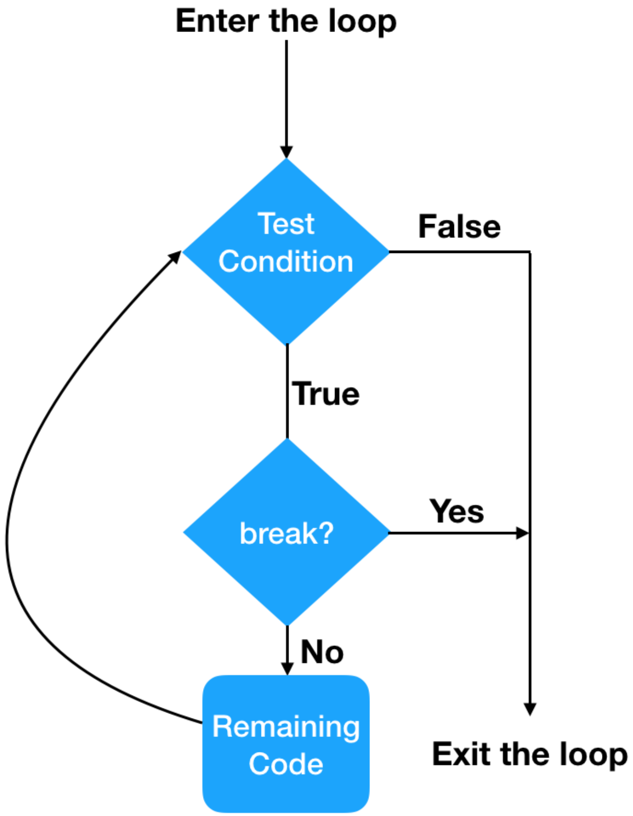
```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

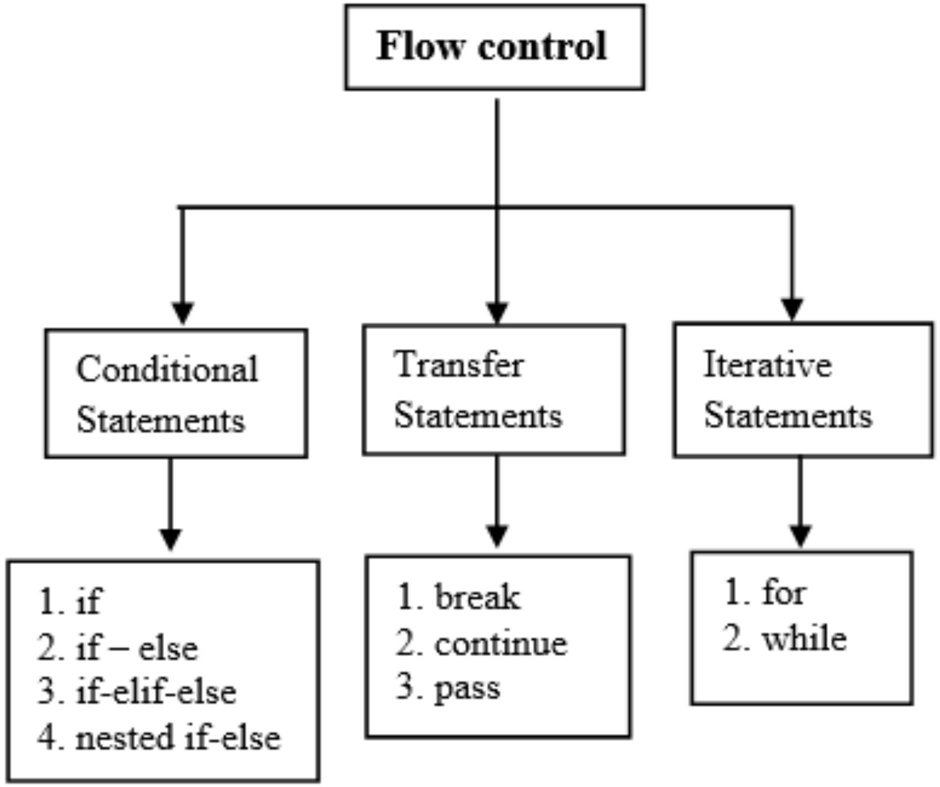
- P03-06-B Asterisks (*)을 사용하여 사용자로부터 입력받은 크기의 직사각형을 출력하는 프로그램을 작성해보세요.
  - input: N (직사각형의 밑변 길이), M (직사각형의 높이 길이) 🡸 사용자로부터 입력
  - output: *로 구성된 N*M 크기의 직사각형

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

Break loop statement

- Python allows programmers to decide whether they want to break out of a loop early or not
  - whether a `for` loop or a `while` loop
  - use break `statement`





Break loop statement

- Typically, if statement is placed on the break so that the break statement is conditionally applied when appropriate

```
print('Only print code if all iterations completed')
num = int(input('Enter a number to check for: '))
for i in range(0, 6):
    if i == num:
        break
    print(i, ' ', end='')
print('Done')
```

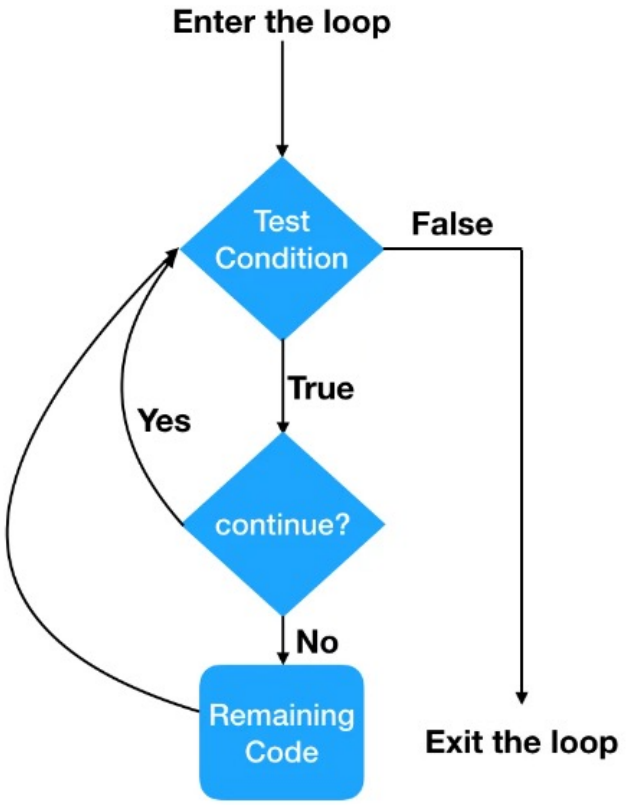
```
Only print code if all iterations completed
Enter a number to check for: 7
0  1  2  3  4  5  Done
```

```
Only print code if all iterations completed
Enter a number to check for: 3
0  1  2  Done
```

  - if the entered value is 7, then all the values in the loop should be printed;
  - else if the value is 3, then only the value 0, 1 2 and 2 will be printed out before loop breaks early

Continue loop statement

- The `continue` statement also affects the flow of control within the lopping constructs `for` and `while`
  - but it does not terminate the whole loop; rather it only terminates the current iteration loop



Continue loop statement

```python
for i in range(0, 10):
    print(i, ' ', end='')

    if i % 2 == 1:
        continue
    print('hey its an even number')
    print('we love even numbers')
print('Done')
```

```
0  hey its an even number
we love even numbers
1  2  hey its an even number
we love even numbers
3  4  hey its an even number
we love even numbers
5  6  hey its an even number
we love even numbers
7  8  hey its an even number
we love even numbers
9  Done
```

Pass statement

- As a placeholder for future code
  - when the pass statement is executed, nothing happens;
  - but, it avoid a syntax error when empty code is not allows

```python
age = 18
if age < 18:
    # TODO: Implement age restriction logic
    pass
else:
    print("You are old enough to vote.")
```

```python
for item in my_list:
    # No action needed for now
    pass
```

```python
def function_that_does_nothing_yet():
    pass
```

```python
class MyEmptyClass:
    pass
```

For loop with else

- A for loop can have an optional else block at the end of the loop
  - else part is executed if and only if all items in the sequence are processed successfully

```
print('Only print code if all iterations completed')
num = int(input('Enter a number to check for: '))
for i in range(0, 6):
    if i == num:
        break
    print(i, ' ', end='')
else:
    print()
    print('All iterations successful')
```

```
Only print code if all iterations completed
Enter a number to check for: 100
0  1  2  3  4  5
All iterations successful
```

For loop with else

- A for loop can have an optional else block at the end of the loop
    - not executed if there are some fails in the loop
        - for loop may fail to process all elements in the loop if for some reason an error occurs (for example by a syntax error) or if you break the loop

```
print('Only print code if all iterations completed')
num = int(input('Enter a number to check for: '))
for i in range(0, 6):
    if i == num:
        break
    print(i, ' ', end='')
else:
    print()
    print('All iterations successful')
```

```
Only print code if all iterations completed
Enter a number to check for: 3
0  1  2
```

Note: Loop variable naming

- Typically, variable names should be meaningful
- The one exception to this rule related to loop variable names used with for loops over ranges
  - very common to find that these loop variables are called ‘i’, ‘j’, etc.
  - you should consider using these variable names in looping constructs,
  - and avoid using them elsewhere

In class practice

- P03-07: 1부터 100까지 정수의 합을 계산하여 출력하는 프로그램을 작성해보세요.
  - 사용자로부터 입력받는 input 없음
  - output: 1부터 100까지의 합
  - note: variable for value of sum should be initialized to 0 first

In class practice

- P03-08 주어진 수의 factorial을 계산하는 프로그램을 작성해보세요.
  - input: 정수 N
  - output: N!
    - if input is 5; factorial of number 5 (often written as 5!) which is 1 * 2 * 3 * 4 * 5 and equals 120
  - not defined for negative numbers' factorial, and 0! is 1
    - if the number is less than 0, return with an error message
    - check to see if the number is 0; print out 1

In class practice

- P03-09 500에서 1000 사이의 정수 중 홀수의 합을 계산하여 출력하는 프로그램을 작성해보세요.
  - variable for value of sum should be initialized to 0 first
  - use `if` statement in `for`/`while` loop statements

```
sum of odd numbers between 500 and 5000 is 187500
```

In class practice

- P03-10 Asterisks (*)을 활용하여 사용자로부터 입력받은 정수에 따라 아래와 같은 역피라미드를 출력해보세요.
    - input: 피라미드의 가장 긴 변의 길이 N
    - output: asterisks으로 구성된 역피라미드
    - example for input value = 5

```
* * * * *
 * * * *
  * * *
   * *
    *
```

- 3 lines: +1 point
- 2 lines: +2 points
- only 1 line: +4 points

4. Error and exception handling

What is exception handling?

- Exception handling (예외 처리)
  - 프로그램 실행 중 발생할 수 있는 오류나 예상치 못한 상황을 처리하기 위한 메커니즘
  - 예외 처리를 통해 프로그램의 안정성과 신뢰성을 확보할 수 있으며, 적절한 대응을 할 수 있음
  - ex) 주민번호 입력란에 한글이 들어간 경우, 영문이름 입력안에 한글이 들어온 경우 등

  - Syntax – 'try-except-finally' statement

```
try:
    # 실행할 코드
except ExceptionType:
    # 예외가 발생했을 때 처리할 코드
finally:
    # 예외 발생 여부와 상관없이 실행되는 부분
```

Examples of exception handling

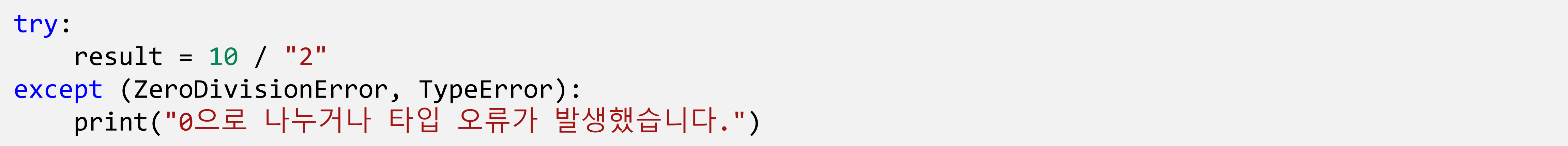
- 단일 예외 처리
  - try-except statement

```python
try:
    # 예외가 발생할 수 있는 코드
    result = 10 / 0
except ZeroDivisionError:
    # ZeroDivisionError 발생 시 실행되는 코드
    print("0으로 나눌 수 없습니다.")
```

```python
try:
    my_list = [1, 2, 3]
    print(my_list[3])
except IndexError:
    print("인덱스 범위를 벗어났습니다.")
```

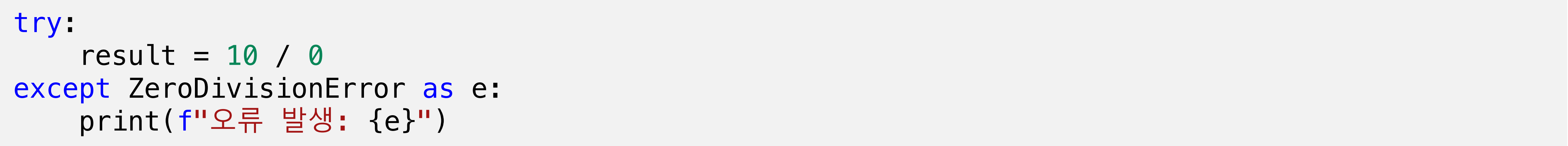
Examples of exception handling

- 여러 예외 동시 처리
  - except문에서 괄호를 사용해서 여러 예외를 동시에 처리



```
try:
    result = 10 / "2"
except (ZeroDivisionError, TypeError):
    print("0으로 나누거나 타입 오류가 발생했습니다.")
```

- 예외의 정보 접근
  - 예외 객체에 접근하여 예외와 관련된 정보를 획득 가능



```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"오류 발생: {e}")
```

Examples of exception handling

- finally
  - 예외 발생 여부와 관계없이 항상 실행되는 코드
  - 주로 자원 해제 등의 정리 작업에 활용

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
finally:
    print("예외 발생 여부와 상관없이 실행됩니다.")
```

Examples of exception handling

- else
  - 예외가 발생하지 않았을 때 실행되는 코드
  - else 블록은 except 블록 다음에 위치해야 함

```python
try:
    result = 10 / 2
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
else:
    print("예외가 발생하지 않았습니다. 결과:", result)
```

Examples of exception handling

- Python에서의 예외
  - Python 표준 라이브러리에 정의된 예외 클래스

BaseException

SystemExit

KeyboardInterrupt

StopIteration

ArithemticError

AttributeError

EOFError

NameError

OSError

TypeError

ValueError

IndexError

ModuleNotFoundError

…

Quiz

- What is the output of the following code?

```
for i in range(20, 10, -3):
    print(i, end=' ')
```

  - a) 19 16 13 10
  - b) 10 13 16 19
  - c) 11 14 17 20
  - d) 20 17 14 11

End of slide