CS112
Introduction to Python Programming

Session 07: Control Flow Statements

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

SUSTech Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

SUSTech Southern University of Science and Technology

- Conditionals
- `if` statement
- `if...else` decision control
- `if...elif...else` decision control
- Nested `if` statement
- Loops
- The `while` loop
- The `for` loop
- List comprehensions using `for` loop
- The `continue` and `break` statements
- `try...except` statement
- `try...except...finally`

Conditionals and loops

- Computer programs are useful for performing repetitive tasks. They can repetitively perform the same calculations with minor, but important, variations over and over again. – Loops
- In the course of doing these repetitive tasks, computers often need to make decisions. We often want to tell the computer ahead of time what to do if it encounters different situations. – Conditions
- Conditionals and loops control the flow of a program. They are essential to performing virtually any significant computational task.
- Python, like most computer languages, provides a variety of ways of implementing loops and conditionals.

Conditionals

- Conditional statements allow a computer program to take different actions based on whether some condition, or set of conditions is true or false. In this way, the programmer can control the flow of a program.
- `if` statement
- `if` ··· `else` decision control
- `if` ··· `elif` ··· `else` decision control
- Nested `if` statement

if statement

- The simplest logical structure is a simple `if` statement, which executes a block of code if some condition is met but otherwise does nothing
- The `if` decision control flow statement starts with `if` keyword and ends with a colon
- The expression in an `if` statement should be a Boolean expression

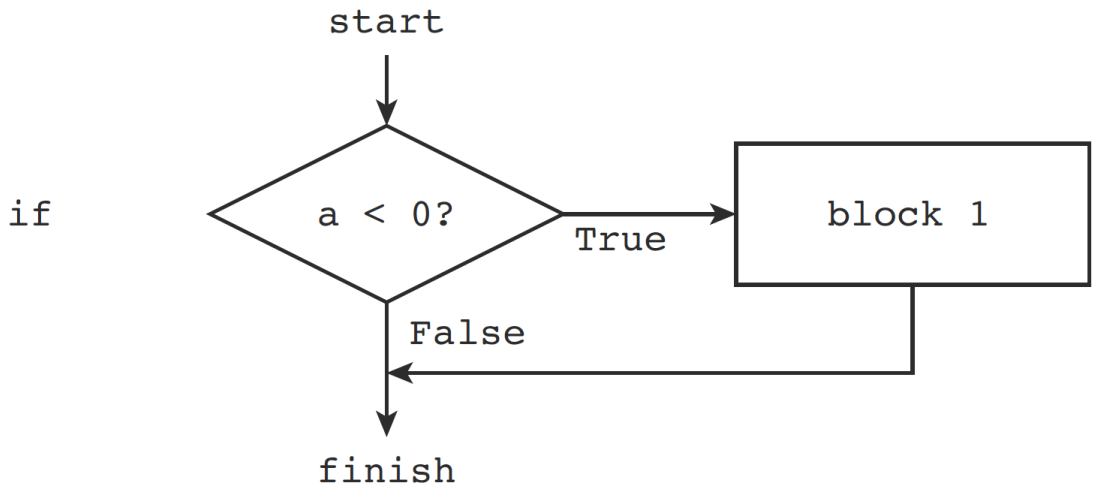
Logical operators

<, <=
>, >=
==
!=
and, or, not

if statement

- The `if` block statements are determined through indentation and the first unindented statement marks the end
- The program below, which demonstrates how to take the absolute value of an integer number:

```
a = int(input("Enter a number: "))
if a < 0:
    a = -a
print("The absolute value is {}".format(a))
```



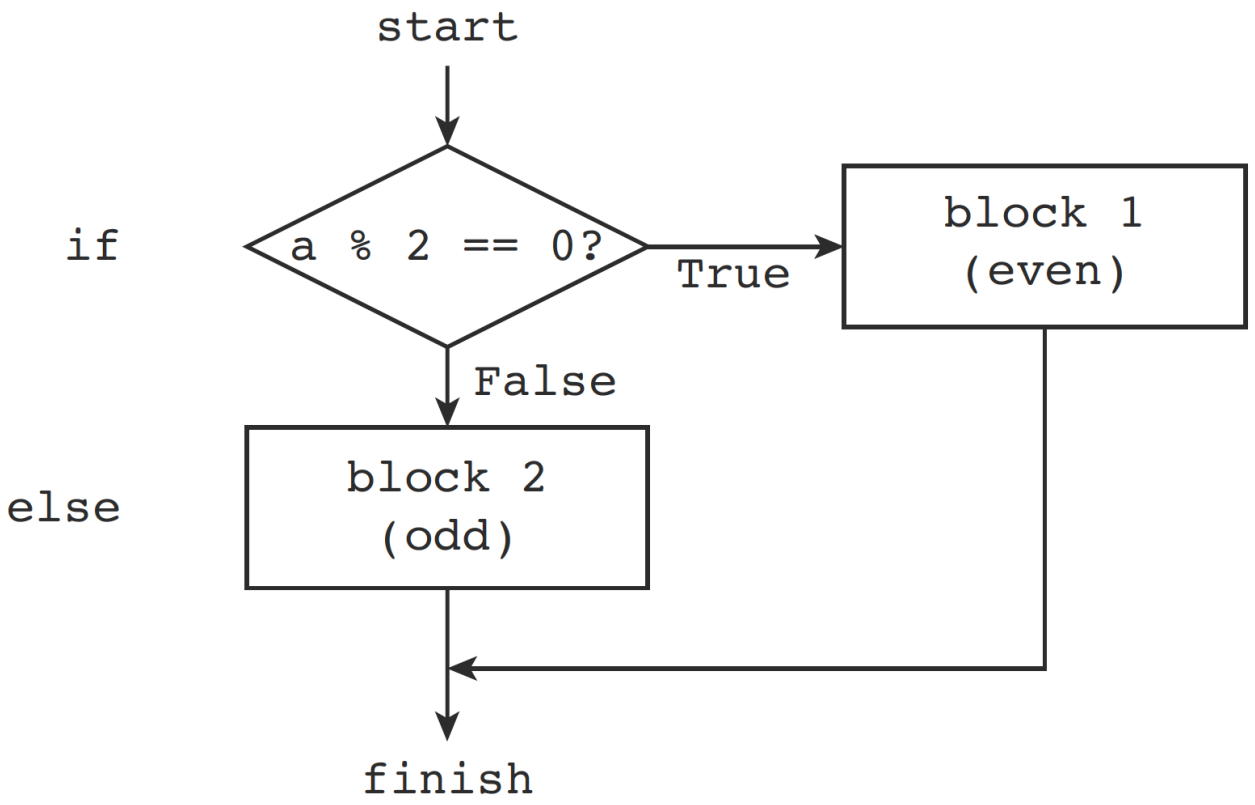
if...else decision control

- An `else` statement does not have any condition
- Statements in the `if` block are executed if the Boolean expression is `True`. Use the optional `else` block to execute statements if the Boolean expression is `False`
- The `if...else` statement allows for a two-way decision
- Indentation is used to separate the blocks. `if` and `else` keywords should be aligned at the same column position

if...else decision control

- The following program testing whether an integer is even or odd provides a simple if···else statement example

```
a = int(input(" Please input an integer: "))
if a % 2 == 0:
    print(f"{a} is Even number")
else:
    print(f"{a} is Odd number")
```



if...elif...else decision control

- The if...elif...else is a multi-way decision control statement
- The optional else statement must always come last, and there can be only one else block
- There can be zero or more elif parts each followed by an indented block
- Only the preceding Boolean expressions evaluated to False, the current statement will be evaluated
- Only the Boolean expression evaluated to True, the statement block will be executed

```
if Boolean_Expression_1:
    statement_1
elif Boolean_Expression_2:
    statement_2
:
:
else:
    statement_last
```

When the block of code in an if or elif statement is only one line long, you can write it on the same line as the if or elif statement

if...elif...else decision control

- Suppose we want to know if the solutions to the quadratic equation

$$ax^2 + bx + c = 0$$

are real, imaginary, or complex for a given set of coefficients *a*, *b*, and *c*. The answer depends on the value of the discriminant *d*:

$$d = b^2 - 4ac$$

The solutions are

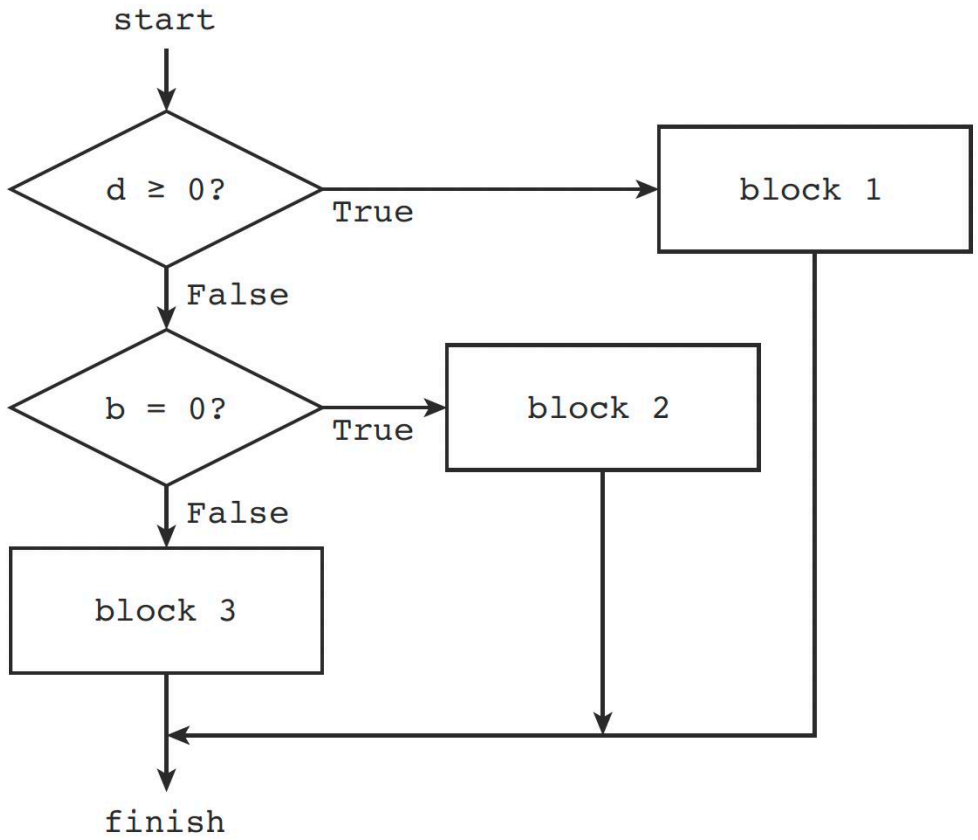
- real if *d* >= 0,
- imaginary if *b* = 0 and *d* < 0, and
- complex if *b* != 0 and *d* < 0

if...elif...else decision control

- The program below implements the above logic in a Python program.

```
a = float(input("What is the coefficient a? "))
b = float(input("What is the coefficient b? "))
c = float(input("What is the coefficient c? "))
d = b*b - 4.*a*c

# block 1
if d >= 0.0:                                          if
    print("Solutions are real")
# block 2
elif b == 0.0:
    print("Solutions are imaginary")                  elif
# block 3
else:
    print("Solutions are complex")                    else
print("Finished")
```



Nested if statement

SUSTech Southern University of Science and Technology

- An if statement that contains another if statement either in its if block or else block is called a nested if statement

```
if Boolean_Expression_1:
    if Boolean_Expression_2:
        statement_1
    else:
        statement_2
else:
    statement_3
```

Loops

- In computer programming a loop is a statement or a block of statements that is executed repeatedly.
- Python has two kinds of loops
  - a `for` loop, and
  - a `while` loop

The while loop

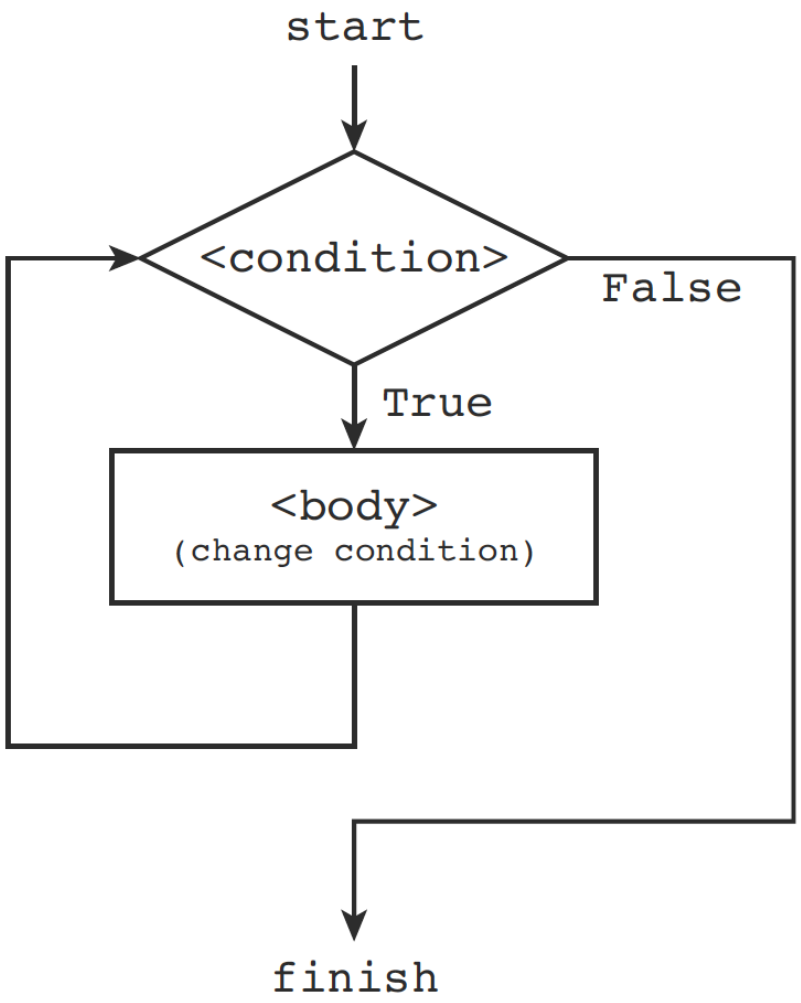
- The `while` loop starts with the `while` keyword and ends with a colon
- The Boolean expression is evaluated before the statements in the `while` loop block is executed
- After each iteration of the loop block, the Boolean expression is again checked, and if it is `True`, the loop is iterated again
- This process continues until the Boolean expression evaluates to `False` and at this point the `while` statement exits:

```
while Boolean_Expression:
    statement(s)
```

The `while` loop

- Write a program to display first 10 numbers using `while` loop starting from 0:

```
In [1]: i = 0
   ...: while i < 10:
   ...:     print(f"The current number is {i}")
   ...:     i += 1
   ...:
The current number is 0
The current number is 1
The current number is 2
The current number is 3
The current number is 4
The current number is 5
The current number is 6
The current number is 7
The current number is 8
The current number is 9
```



The `while` loop

- Suppose you want to calculate all the Fibonacci numbers smaller than 1000. The Fibonacci numbers are determined by starting with the integers 0 and 1. The next number in the sequence is the sum of the previous two. So, starting with 0 and 1, the next Fibonacci number is 0 + 1 = 1, giving the sequence 0;1; 1. Continuing this process, we obtain 0;1;1;2;3;5;8; ::: where each element in the list is the sum of the previous two:

```
In [6]: x, y = 0, 1
   ...: while x < 1000:
   ...:     print(x, end=" ")
   ...:     x, y = y, x+y
   ...:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

The end=' ' argument causes a space to be printed out between each value of i instead of the default new line character \n.

The for loop

- The `for` loop starts with the `for` keyword and ends with a colon
- The first item in the sequence gets assigned to the iteration variable *iteration_variable*
- Then the statement block is executed
- This process of assigning items from the sequence to the *iteration_variable* and then executing the statements continues until all the items in the sequence are completed

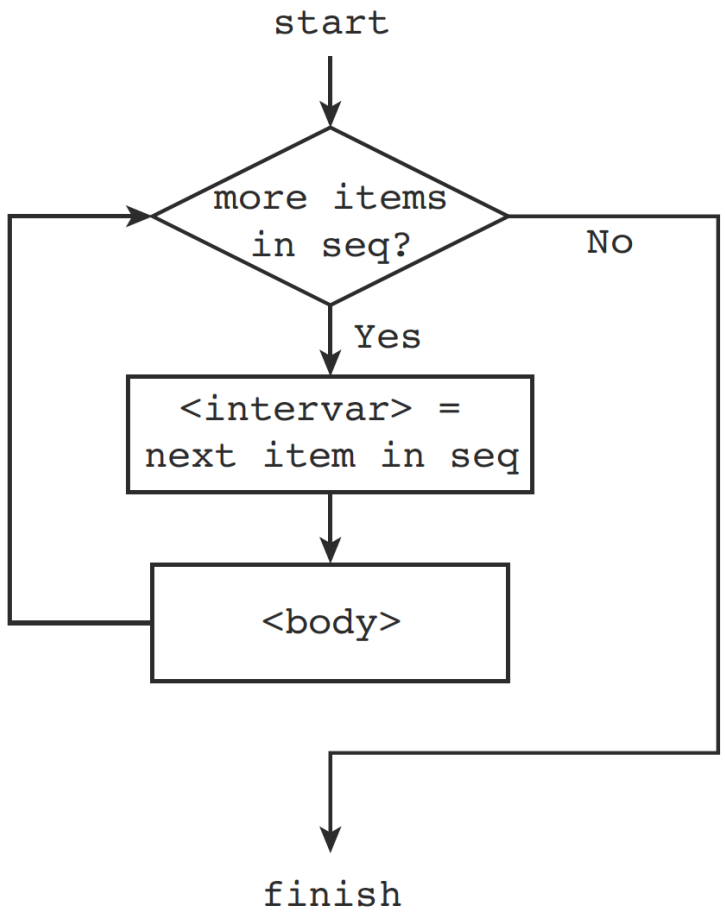
```
for iteration_variable in sequence:
    statement(s)
```

```
for each_char in "Blue":
    print(f"Iterate through character {each_char} in the string 'Blue'")
```

The for loop

- In the following example, the loop cycles through all the elements of the list, and then moves on to the code that follows the for loop and prints “All done.”

```
In [2]: for dogname in ["Molly", "Max", "Buster", "Lucy"]:
   ...:     print(dogname)
   ...:     print(" Arf, arf!")
   ...: print("All done.")
Molly
 Arf, arf!
Max
 Arf, arf!
Buster
 Arf, arf!
Lucy
 Arf, arf!
All done.
```



The for loop

SUSTech Southern University of Science and Technology

- The `range()` function generates a sequence of numbers which can be iterated through using `for` loop
- The syntax for `range()` function is

```
range([start ,] stop [, step])
```

e.g.,

```
for i in range(2, 5):
    print(f"{i}")
```

```
In [3]: s = 0
   ...: for i in range(1, 100, 2):
   ...:     print(i, end=' ')
   ...:     s = s+i
   ...: print('\n{}'.format(s))
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
93 95 97 99
2500
```

List comprehensions using for loop

- List comprehensions are a special feature of core Python for processing and constructing lists. We introduce them here because they use a looping process

```
>>> A = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
>>> A
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> diag = []
>>> for i in [0, 1, 2]:
...     diag.append(A[i][i])
>>> diag
[1, 5, 9]
```

- List comprehensions provide a simpler, cleaner, and faster way to build a list of the diagonal elements of A

```
>>> diagLC = [A[i][i] for i in [0, 1, 2]]
>>> diagLC
[1, 5, 9]
```

- Notice here how y serves as a dummy variable accessing the various elements of the list diagLC

```
>>> [y*y for y in diagLC]
[1, 25, 81]
```

List comprehensions using `for` loop

- Extracting a row from a 2-dimensional array such as A is quite easy. For example the second row is obtained quite simply in the following fashion:

```
>>> A[1]
[4, 5, 6]
```

- Obtaining a column is not as simple, but a list comprehension makes it quite straightforward:

```
>>> c1 = [a[1] for a in A]
>>> c1
[2, 5, 8]
```

a → list

- Another, slightly less elegant way to accomplish the same thing is

```
>>> [A[i][1] for i in range(3)]
[2, 5, 8]
```

- extract all the elements of a list that are divisible by three

```
>>> y = [-5, -3, 1, 7, 4, 23, 27, -9, 11, 41]
>>> [x for x in y if x%3==0]
[-3, 27, -9]
```

The continue and break statements

- The `break` and `continue` statements provide greater control over the execution of code in a loop
- Whenever the `break` statement is encountered, the execution control immediately jumps to the first instruction following the loop
- To pass control to the next iteration without exiting the loop, use the `continue` statement
- Both `continue` and `break` statements can be used in `while` and `for` loops

The continue and break statements

- Examples of the continue and break statements:

```
In [9]: n = 10
   ...: while n > 0:
   ...:     n -= 1
   ...:     if n == 5:
   ...:         continue
   ...:     print(f"The current value is {n}")
   ...:
The current value is 9
The current value is 8
The current value is 7
The current value is 6
The current value is 4
The current value is 3
The current value is 2
The current value is 1
The current value is 0
```

```
In [10]: n = 10
   ...: while n > 0:
   ...:     n -= 1
   ...:     if n == 5:
   ...:         break
   ...:     print(f"The current value is {n}")
   ...:
The current value is 9
The current value is 8
The current value is 7
The current value is 6
```

The continue and break statements

- Examples of the continue and break statements:

```
In [7]: current_number = 0
   ...:
   ...: while current_number < 10:
   ...:     current_number += 1
   ...:     if current_number % 2 == 0:
   ...:         continue
   ...:
   ...:     print(current_number)
   ...:
1
3
5
7
9
```

```
In [8]: current_number = 0
   ...:
   ...: while current_number < 10:
   ...:     current_number += 1
   ...:     if current_number % 2 == 0:
   ...:         break
   ...:
   ...:     print(current_number)
   ...:
1
```

break jump out of the loop while continue skip the current iteration

try...except statement

- There are at least two distinguishable types of errors: Syntax Errors and Exceptions
- Syntax Errors:

```
while True
    print("Hello World)

while True
               ^
SyntaxError: invalid syntax
```

try...except statement

- Exceptions: errors detected during execution

```
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

```
In [1]: try:
   ...:     print(5/0)
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero!")
   ...:
You can't divide by zero!
```

try...except...finally

- Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs:

```
try:
    statement_1
except Exception_Name_1:
    statement_2
except Exception_Name_2:
        statement_3
        .
        .
        .
else:
    statement_4
finally:
    statement_5
```

- A `try` block consisting of one or more statements is used by Python programmers to partition code that might be affected by an exception
- The associated `except` blocks are used to handle any resulting exceptions thrown in the `try` block
- only one except block is executed for each exception that is thrown
- A `finally` block is always executed before leaving the try statement, whether an exception has occurred or not

try...except...finally

- Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs:

```
while True:
    try:
        number = int(input("Enter a number: "))
        print(f"The number entered is {number}")
        break
    except ValueError:
        print("Oops! That's not a valid number. Try again…")
```

try...except...finally


```
In [4]: a = int(input("Enter a number:"))
Enter a number:5
In [5]: try:
   ...:     b = 100/a
   ...: except TypeError:
   ...:     print("unsupported operand types")
   ...: except ZeroDivisionError:
   ...:     print("can't divide by zero")
   ...: finally:
   ...:     print("Always run this block")
   ...:
Always run this block
```

```
In [6]: a = '5'
In [7]: try:
   ...:     b = 100/a
   ...: except TypeError:
   ...:     print("unsupported operand types")
   ...: except ZeroDivisionError:
   ...:     print("can't divide by zero")
   ...: finally:
   ...:     print("Always run this block")
   ...:
unsupported operand types
Always run this block
```

```
In [8]: a = 0
In [9]: try:
   ...:     b = 100/a
   ...: except TypeError:
   ...:     print("unsupported operand types")
   ...: except ZeroDivisionError:
   ...:     print("can't divide by zero")
   ...: finally:
   ...:     print("Always run this block")
   ...:
can't divide by zero
Always run this block
```

The `finally` keyword is used to create finally block, that executes all the statements written in this block, without caring whether an exception is raised or not.

Exercise

Exercise 1

- Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error：

Score	Grade
≥ 0.9	A
≥ 0.8	B
≥ 0.7	C
≥ 0.6	D
< 0.6	F

Exercise 2

- Write a program to check if a given year is a leap year
- All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only if it is perfectly divisible by 400

Exercise 3

- Write a program to find the average of *n* natural numbers (from 1 to *n*) using `while` loop, where *n* is the input from the user:

Exercise 4

- Write a program to repeatedly check for the largest user input positive number until the user enters “done”:

Exercise 5

- Write a program to find the sum of all odd and even numbers up to a number specified by the user:

Exercise 6

- Print integer numbers using while loop, and stop when the number reaches 5:

Exercise 7

- Write a program to check for ZeroDivisionError exception when dividing two user input numbers and giving the result: