Python Programming

Conditions and Iterations

Prof. Chang-Chieh Cheng
Information Technology Service Center
National Yang Ming Chiao Tung University

Boolean Value

- **`False`**
  - Zero number
  - None
  - Empty
- **`True`**
  - Non-false value

Comparison Operators

- `>` Greater than
  - `x > y` is `True` if `x` is grater than `y`; otherwise, it returns `False`.
- `<` Less than
- `>=` Greater than or equal
- `<=` Less than or equal
- `==` Equal
- `!=` Not equal

```
x = 1
y = 2
print(x > y)   # False
print(x >= y)  # False
print(x < y)   # True
print(x <= y)  # True
print(x == y)  # False
print(x != y)  # True
```

Comparison Operators

- `==` and `!=` can be used to check if two lists have the same contents.

```
L1 = [1, 2, 3]
L2 = [3, 1, 2]
L3 = [1, 2, 3]
L4 = [1, 2, 3, 4]
print(L1 == L2) # False
print(L1 == L3) # True
print(L1 == L4) # False
```

- `==` and `!=` can also be used to check if two strings are the same.

```
s1 = 'Hello'
s2 = 'hello'
s3 = 'Hello'
s4 = 'Hello!'
print(s1 == s2) # False
print(s1 == s3) # True
print(s1 == s4) $ False
```

Logical Operators

- and

x \ y	True	False
True	True	False
False	False	False

- or

x \ y	True	False
True	True	True
False	True	False

- not
  - not True ➔ False
  - not False ➔ True

```
x = 5
print(x > 0 and x < 10)          # True
print(x > 0 and x < 5)           # False
print(x > 0 or x < 5)            # True
print(x < 0 or x > 5)            # False
print(not(x < 0 or x > 5) )      # True
```

Membership Operators

- `in`
  - Let `L` be a data set.
  - `x` **`is in`** `L` results `True` if `x` belongs to `L`.
- `not in`
  - `x` **`not in`** `L` results `True` if `x` does not belong to `L`.

```
L = [1, 2, 3, 4, 5]
print( 3 in L )                 # True
print( 6 in L )                 # False

print( 3 not in L )             # False
print( 6 not in L )             # True
```

Identity Operators

- `is`
  - Given two objects, `x` and `y`.
  - `x is y` results `True` if `x` and `y` refer to the same object.
- `is not`
  - `x is not y` results `True` if `x` and `y` refer to two different objects.

```
L1 = [1]
L2 = [1]
print( L1 is L1 )              # True
print( L1 is L2 )              # False
print( L1 is not L1 )          # False
print( L1 is not L2 )          # True
```

`if` statement

- Syntax:

```
if condition:
    indented_statement_block
```

- For example, check a number is even.

```
x = int(input('Input an integer:'))
if x % 2 == 0:
    print(x, 'is even')
```

`if-else` statement

- Syntax:

```
if condition:
    indented_statement_block
else:
    indented_statement_block
```

- For example, check a number is even or odd.

```
x = int(input('Input a integer:'))
if x % 2 == 0:
    print(x, 'is even')
else:
    print(x, 'is odd')
```

`if-elif-else` statement

- Syntax:

```
if condition1:
    indented_statement_block1
elif condition2:
    indented_statement_block2
elif condition3:
    indented_statement_block3
...
else:
    indented_statement_blockE
```

  - Notice that else part must be the final part.

- For example, classify a score.

```
x = int(input('Input a score:'))
if x >= 90:
    print(x, 'is excellent!')
elif x >= 80:
    print(x, 'is good!')
elif 80 > x >= 60:
    print(x, 'is ok!')
else:
    print(x, 'is failed!')
```

Nested `if-elif-else` statement

```
x = int(input('Input a score:'))
if x >= 60:
    print('Pass!')
    if x >= 90:
        print(x, 'is excellent!')
    elif x >= 80:
        print(x, 'is good!')
    else:
        print(x, 'is ok!')
else:
    if x >= 50:
        print(x, 'still has a chance.')
    else:
        print(x, 'is failed!')
```

Conditional Expressions

- Syntax:

```
true_value if condition else false_value
```

- Example:

```
x = int(input('Input a number:'))
print('Pass!') if x >= 60 else print('Failed!')
```

```
x = int(input('Input a score:'))
x = 100 if x > 90 else x
```

Exercise

- Use `input()` to design a program that allows the user to enter a text `t`.
- `print('%s is a palindrome.'%t)` If `t` is a palindrome that reads the same backwards as forwards, such as `'abba'`, `'tenet'`, and `'1234321'`.
- Otherwise, `print('%s is not a palindrome.'%t)`.
- DO NOT use any iterative statement.

`while` statements

- **Syntax**

```
while condition:
    indented_statement_block
```

  - Two steps:
    1. If `condition` is True then execute `indented_statement_block` once; Otherwise, stop the `while` statement.
    2. Go back to step 1.
- For example, print the numbers from n to 1.

```
n = int(input('Input a positive integer: '))
while n > 0:
    print(n)
    n = n - 1
```

- Let's try it, print the numbers from 1 to n

`while` statements

- For example, n!, the factorial of n.
  - $n! = 1 \times 2 \times \cdots \times (n-1) \times n$
  - 0 * 1 * n * (n-1) * (n-2) *..... *2*1

```
n = int(input('Input a positive integer: '))
fac = 1
while n > 0:
    fac = fac * n
    n = n - 1
print('n! is', fac)
```

`while` statements

- Let's try it
  - given an even integer *n* and *n* > 1, design a while loop to compute α and β, where

$$\alpha = 2 \times 4 \times 6 \times \cdots \times (n)$$
$$\beta = 1 \times 3 \times 5 \times \cdots \times (n-1)$$

  - For example, if *n* is 10 then

$$\alpha = 2 \times 4 \times 6 \times 8 \times 10 = 3840$$
$$\beta = 1 \times 3 \times 5 \times 7 \times 9 = 945$$

`while` statements

- Be careful when comparing a floating-point number in the condition.
- For example, collecting 10 integers from 0 to 9

```
L = []
n = 0
while n < 10:
  L += [n]
  n += 1
print(len(L))
print(L)
```

- Collecting 10 floating-point numbers from 0.0 to 0.9

```
L = []
n = 0.0
while n < 1.0:
  L += [n]
  n += 0.1
print(len(L))
print(L)
```

`while` statements

- Multiple-layer `while`
- For example, print a 2D array

```
rows, columns = 3, 4
r = 0
while r < rows:
  c = 0
  L = []
  while c < columns:
    L += [r * columns + c]
    c += 1
  print(L)
  r += 1
```

- Let's try it
  - Modify the above code, such that the results will be

```
[[0, 1, 2, 3],
 [10, 11, 12, 13],
 [20, 21, 22, 23]]
```

`while` statements

- Let's try it
  - Design a two-layer `while` statement to generate the following result:

```
OXOXO
XOXOX
OXOXO
XOXOX
OXOXO
```

  - Furthermore, can you generate any size of the above pattern? For example, 7 x 7, 10 x 10 or 7 x 10.

`while` statements

- Input loop
  - For example, sum of non-negative numbers

```
n = 0
sum = 0
while n >= 0:
    n = int(input('Input a non-negative integer: '))
    if n >= 0:
        sum = sum + n
print('sum is', sum)
```

  - String appending

```
s = ''
sOut = ''
while s != '.':
    s = input('Input a word: ')
    if s == '.':
        sOut = sOut + s
    else:
        sOut = sOut + ' ' + s
print(sOut)
```

`while` statements

- Infinite loop

```
m = 0
while True:
    n = int(input('Input a positive integer: '))
    m += n
    print(m)
```

  - How to stop it?

`while` statements

- `break`
  - Escape the current while loop a level without any condition.

```
L = []
while True:
  n = int(input('Input a positive integer: '))
  if n <= 0:
    print(n, '<= 0')
    break
  L += [n]
  print(L)
print(L, sum(L))
```

`while` statements

- Check whether a positive integer is prime
  - Naive algorithm:
    - An integer is a prime if it is greater than 1 and has no positive divisors other than 1 and itself.
    - For example, 5 is a prime because it can't be divided by 2, 3, and 4; 6 is not a prime because it can be divided by 2.

```
n = 62329357
k = 2
while k < n:
    if n % k == 0:
        break
    k += 1
if k == n:
    print(n, 'is a prime number.')
else:
    print(n, 'is not a prime number.')
```

`while` statements

- `continue`
  - Skip an iteration

```
L = []
while True:
  n = int(input('Input a positive integer: '))
  if n <= 0:
    print(n, '<= 0')
    continue
  L += [n]
print(L, sum(L))
```

  - Let's try it
    - Using **`break`** to safely terminate the above program when the input string cannot be converted into an integer.

Exercise 1

- Given two integers, `rows` and `columns`, to indicate a 2D array of **`m` prime numbers** from 2 in ascending order, where `m = rows * columns`.
- Design a program to print this 2D table.
- For example, `rows` and `columns`, are 3 and 4, respectively; the result will be

```
[[2, 3, 5, 7],
 [11, 13, 17, 19],
 [23, 29, 31, 37]]
```

`for` statements

- Syntax

```
for item in data_sequence:
    indented_statement_block
```

  - Each iteration accesses an item in `data_sequence` (a list or a string), in the order that it appear in the sequence.

  - Example:

```
L = ['dog', 'cat', 'bird']
for pet in L:
    print(pet)
```

`for` statements

- Let's try it
  - Given a list `L` that contains `n` integers.
  - Design a `for` loop to create two lists;
    - `L1` contains all **odd** integers of `L`, and
    - `L2` contains all **even** integers of `L`.
  - For example, `L = [8, 9, 2, 5, 6, 4, 7]`
    - `L1` will be `[9, 5, 7]`
    - `L2` will be `[8, 2, 6, 4]`

`for` statements

- `for` and `while`

```
L = [100, 97, 13, 100, 56, 97, 17, 32, 97, 56]
P = []
NP = []
for n in L:
    k = 2
    while k < n:
        if n % k == 0:
            break
        k += 1
    if k == n:
        P += [n]
    else:
        NP += [n]
print('Prime numbers:', P)
print('Non-prime numbers:', NP)
```

`for` statements

- `for`, `while`, `break`, and `continue`

```
L = [100, 97, 13, 100, 56, 97, 17, 32, 97, 56]
P = []
NP = []
for n in L:
    if n in P or n in NP:
        continue
    k = 2
    while k < n:
        if n % k == 0:
            break
        k += 1
    if k == n:
        P += [n]
    else:
        NP += [n]
print('Prime numbers:', P)
print('Non-prime numbers:', NP)
```

`for` statements

- `range` function
  - Generating a sequence of numbers that over a specified range
  - Three usages
    - `range(stop)` ➔ 0 to `stop` - 1
    - `range(start, stop)` ➔ `start` to `stop` - 1
    - `range(start, stop, step)`
      If step > 0 ➔ `start, start + step, …, (start + i * step) < stop`
      If step < 0 ➔ `start, start + step, …, (start + i * step) > stop`
    - `start, stop,` and `step` must be integers
    - step must be non-zero
  - Example:

```
L = list(range(5))
print(L)                # 0, 1, 2, 3, 4
L = list(range(6, 10))
print(L)                # 6, 7, 8, 9
L = list(range(0, 10, 2))
print(L)                # 0, 2, 4, 6, 8
L = list(range(0, -10, -2))
print(L)                # 0, -2, -4, -6, -8
```

`for` statements

- `for` statement with `range` function

```
for x in range(-100, 100, 10):
    print(x)
```

- Range normalization
  - The following code generates a number sequence from -1.0 to 0.99

```
L = []
for x in range(-100, 100):
    L.append(x * 0.01)
print(L)
```

`for` statements

- Using `for` statement to access each item of a list

```
L = [1, 2, 3, 4, 5]
for i in range(len(L)):
    L[i] *= 10
for i in range(len(L)):
    print(L[i])
```

`for` statements

- Let's try it
  - [0, 1] Normalization:
    - Given a list `L` that contains `N` numbers
    - Let `min` is the minimum of `L`, and `max` is the maximum of `L`
    - Then, each number `x` in `L` =
      - `(x - min) / (max - min)`
    - For example, `L = [-10, -2, 0 , 3, 4]`, the results is `[0.0, 0.571, 0.714, 0.929, 1.0]`

sorting

- `sorted` function
  - Ascending sorting

```
L1 = [5, 4, 2, 3, 1]
L2 = sorted(L1)
print(L1)       # [5, 4, 2, 3, 1]
print(L2)       # [1, 2, 3, 4, 5]
```

```
Ls1 = ['cat', 'mouse', 'pig', 'dog', 'bird']
Ls2 = sorted(Ls1)
print(Ls1)      # ['cat', 'mouse', 'pig', 'dog', 'bird']
print(Ls2)      # ['bird', 'cat', 'dog', 'mouse', 'pig']
```

  - The order of characters
    - Based of **ASCII** (American Standard Code for Information Interchange)
    - symbols(`!#$..`) <
      digits(`0-9`) <
      upper-case alphabets(`A-Z`) <
      lower-case alphabets(`a-z`) <
      `{, |, }, ~`

sorting

- `sorted` function
  - Descending sorting

```
L1 = [5, 4, 2, 3, 1]
L2 = sorted(L1, reverse = True)
print(L1)       # [5, 4, 2, 3, 1]
print(L2)       # [5, 4, 3, 2 ,1]
```

```
Ls1 = ['cat', 'mouse', 'pig', 'dog', 'bird']
Ls2 = sorted(Ls1 , reverse = True)
print(Ls1)      # ['cat', 'mouse', 'pig', 'dog', 'bird']
print(Ls2)      # ['pig', 'mouse', 'dog', 'cat', 'bird']
```

sorting

- `sort` method
  - Ascending sorting

```
L1 = [5, 4, 2, 3, 1]
L1.sort()
print(L1)       # [1, 2, 3, 4, 5]
```

```
Ls1 = ['cat', 'mouse', 'pig', 'dog', 'bird']
Ls1.sort()
print(Ls1)      # ['bird', 'cat', 'dog', 'mouse', 'pig']
```

sorting

- `sort` method
  - Descending sorting

```
L1 = [5, 4, 2, 3, 1]
L1.sort(reverse = True)
print(L1)       # [5, 4, 3, 2 ,1]
```

```
Ls1 = ['cat', 'mouse', 'pig', 'dog', 'bird']
Ls1.sort(reverse = True)
print(Ls1)      # ['pig', 'mouse', 'dog', 'cat', 'bird']
```

sorting

- Let's try it
  - Input a text `s`
  - Change this text to a list of characters , `L`
  - Sort `L` by order of ASCII
  - For example
    - `s = 'I am a smart guy!'`
    - `L` will be:
      - `[' ', ' ', ' ', ' ', '!', 'I', 'a', 'a', 'a', 'g', 'm', 'm', 'r', 's', 't', 'u', 'y']`

sorting

- Let's try it
  - Input a positive integer `n`
  - Split all digits to a list of integers, `L`
  - Sort `L` by order of numbers
  - Then, transform L to an integer m
  - Print n + m
    - For example
      - `n = 8573`
      - `L` will be:
        - `[3, 5, 7, 8]`
      - then, `m = 3578`
      - `n + m = 12151`

Exercise

- Given a list `A` with `N` numbers
  - Finding the **maximum**, **minimum** and **median**
    - The index of median in `N` sorted numbers is `int(N/2)`
  - Calculating the **standard deviation of median** by the following equation

$$s_m = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(A[i] - \boldsymbol{m})^2}$$

    - where `m` is the median
  - The error of your results should be in ±0.0001.
  - For example, `A = [10, 5, 1, 6, 9, 2, 6, 9, 10, 8]`, s_m will be about `3.5277`