Python Programming

Functions and Modules

Prof. Chang-Chieh Cheng

Information Technology Service Center

National Chiao Tung University

Functions

- Some pieces of code are useful and can be used again in the other places
- For example, computing the average of a list

```
L1 = [4, 5, 2, 1, 9]
avg1 = sum(L1) / len(L1)
print(avg1)
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
avg2 = sum(L2) / len(L2)
print(avg2)
```

- Do you need to write such reusable code again for using in next time?
  - No
  - Make it as a function!
  - Then call the function when you want to use it

Functions

- Syntax of a function definition

```
def function_name(parameter):
    function_code_block
```

`def`: abbreviation of **define**

- Output the result of a function
  - `return value`
  - For example

```
def avg(L):                          # function definition
    return sum(L) / len(L)
                                     # L is the parameter of avg

L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
print(avg(L1))                       # function calling
print(avg(L2))                       # function calling

# L1 and L2 are arguments for function calling of avg
```

Functions

- Let's try it
  - Design a function named `median` that can find the median from a list.
  - So, the following program can be executed correctly.

```
L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
print(median(L1))                    # 4
print(median(L2))                    # 0.4
```

Functions

- No-return function
- For example, printing each item of a list with index

```
def printList(L):
    i = 0
    n = len(L)
    while i < n:
        print('[', i, ']', L[i], sep = '')
        i += 1

L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
printList(L1)
printList(L2)
```

Functions

- Function with multiple parameters
- An example, element-wise addition for two lists

```
def sumList(L1, L2):
    i = 0
    Lr = []
    while i < len(L1) and i < len(L2):
        Lr.append(L1[i] + L2[i])
        i += 1
    return Lr

L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
L3 = sumList(L1, L2)
printList(L3)          # [4.4, 5.5, 2.2, 1.1, 9.9]
```

Functions

- Let's try it
  - append two parameters to `sumList`, `start` and `stop`, to indicate a data range of `L1` and `L2`.
  - Then, `sumList` returns a list that contains the result of element-wise addition of the specified range of `L1` and `L2`.
  - Try to let the following program can be executed correctly.

```
L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
L3 = sumList(L1, L2, 0, 5)
print(L3)                    # [4.4, 5.5, 2.2, 1.1, 9.9]
L4 = sumList(L1, L2, 2, 4)
print(L4)                    # [2.2, 1.1]
L5 = sumList(L1, L2, 1, 4)
print(L5)                    # [5.5, 2.2, 1.1]
```

- The answer is in the next page.

Functions

- Default arguments

```
def sumList(L1, L2, start = 0, stop = 0):
    if stop <= start:                    # stop must > start
        stop = min(len(L1), len(L2))
    i = start
    Lr = []
    while i < stop:
        Lr.append(L1[i] + L2[i])
        i += 1
    return Lr

L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
L3 = sumList(L1, L2)
printList(L3)
L4 = sumList(L1, L2, 2)
printList(L4)
```

Functions

- Keyword argument
  - Specify an argument by its parameter name

```
def sumList(L1, L2, start = 0, stop = 0):
    if stop <= start:                    # stop must > start
        stop = min(len(L1), len(L2))
    i = start
    Lr = []
    while i < stop:
        Lr.append(L1[i] + L2[i])
        i += 1
    return Lr

L1 = [4, 5, 2, 1, 9]
L2 = [0.4, 0.5, 0.2, 0.1, 0.9]
L3 = sumList(L1, L2, start = 1, stop = 4)
printList(L3)
L4 = sumList(L1, L2, stop = 3)
printList(L4)
```

Functions

- Let's try it
  - Design a function, `leftpad(s, n, c)`
    - `s` and c are strings, n is a positive integer
  - `leftpad` can padding a series of c to the left side of `s` such that the length of padded s is n.
  - Try to let the following program can be executed correctly.

```
s = '1.234'
print(leftpad(s, 8, 'X'))     # xxx1.234
print(leftpad(s, 8))          # 0001.234
print(leftpad(s, 15, 'ABCD')) # CDABCDABCD1.234
print(leftpad(s, 0))          # 1.234
print(leftpad(n = 7, c = '@', s = s))# @@1.234
```

  - You can use range access in a string

```
s = 'ABCDEF'
print(s[1:3])  # BC
print(s[:3])   # ABC
print(s[2:])   # CDEF
```

Functions

- Arbitrary arguments

```
def sumA(*A):
    print(type(A))     # tuple
    s = 0
    for x in A:
        s += x
    return s

print(sumA(1, 2, 3))     # 6
print(sumA(1, 2, 3, 4)) # 10
print(sumA())            # 0
```

```
def linear(a, b = 0, *A):
    L = []
    for x in A:
        L.append(a * x + b)
    return L

print(linear(10, 5, 1, 2, 3))
print(linear(10, 5))
print(linear(10))
```

Functions

- Keyword arguments

```
def maxScore(**Scores):
    print(type(Scores))
    name = ''
    score = 0;
    for key in Scores:
        if Scores[key] > score:
            name = key
            score = Scores[key]
    print(name, score)

maxScore(James = 90, Mary = 95, Bill = 86)
maxScore(Wilson = 80, Marks = 65, Emily = 81, Tina = 100)
```

```
def setA(A, **value):
    for key in value:
        i = int(key[1:])
        A[i] = value[key]
A = [0, 0, 0, 0, 0, 0, 0]
setA(A, a0 = 100, a2 = 30, a5 = 10)
print(A)
# What if setA(A, 0 = 100, 2 = 30, 5 = 10)?
```

Functions

- In Python, the values of arguments are copied to corresponded parameters➔ pass-by-value style
- For example

```
def swap(a, b):
    t = a
    a = b
    b = t       # it swaps a and b rather than swaps x and y.

x = 10
y = 20
swap(x, y)      # x's value ➔ a; y's value ➔ b
print(x, y)     # still 10, 20
```

Functions

- Multiple returns

```
def swap(a, b):
    return b, a

x = 10
y = 20
x, y = swap(x, y)
print(x, y)             # 20, 10

s1 = 'xyz'
s2 = 'abc'
s1, s2 = swap(s1, s2)
print(s1, s2)           # abc xyz

L1 = [1, 2, 3]
L2 = ['dog', 'cat']
L1, L2 = swap(L1, L2)
print(L1)               # ['dog', 'cat']
print(L2)               # [1, 2, 3]
```

Functions

- Multiple returns

```
def swap(a, b):
    return b, a

x = 10
y = 20
x = swap(x, y)
print(x)        # (20, 10) ➔ this is a tuple of two elements
print(y)        # 20

s1 = 'xyz'
s2 = 'abc'
s1 = swap(s1, s2)
print(s1)       # (abc, xyz)
print(s2)       # abc

L1 = [1, 2, 3]
L2 = ['dog', 'cat']
L1 = swap(L1, L2)
print(L1)       # (['dog', 'cat'], [1, 2, 3])
print(L2)       # ['dog', 'cat']
```

A tuple is a data sequence that contains several data element of different data type

Functions

- Let's try it
  - Design a function: `feature(L)`
    - where `L` is a list
  - `feature` returns three numbers: median, minimum, and maximum.
  - Try to let the following program can be executed correctly

```
L1 = [4, 5, 9, 1, 2]
L2 = [0.2, 0.9, 0.1, 0.5, 0.4]
med, min, max = feature(L1)
print(med, min, max)          # 4 1 9
med, min, max = feature(L2)
print(med, min, max)          # 0.4 0.1 0.9
```

Functions

- Throwaway objects
  - For any object accessing, an underline symbol _ means that the accessing of this object will be ignored.

```
L1 = [4, 5, 9, 1, 2]
L2 = [0.2, 0.9, 0.1, 0.5, 0.4]
med, min, max = feature(L1)
print(med, min, max)          # 4 1 9
med, _, max = feature(L2)
print(med, min, max)          # 0.4 1 0.9
```

```
for _ in range(5):
    print('*')
```

```
import random
def randomlist(n):
    return [random.random() for _ in range(n)]

print(randomlist(5))
```

Functions

- Local objects
  - All parameters are local objects

```
def func1(n):  # n in the func1
    n *= 10
    return n

n = 10         # n in the main program
m = func1(n)
print(n)       # still 10
print(m)       # 100

m = func1(n)
print(n)       # still 10
print(m)       # still 100
```

Functions

- Local objects
  - All objects initialized in function are local

```
def func2(x):
    n = x                # n in the func2
    return n

n = 10
m = func2(5)
print(n)        # still 10
print(m)        # 5
```

Functions

- Global objects

```
def func3(x):
    global n   # n is a global object
    n *= x
    return n

n = 10
m = func3(n)   # func3 has side effect
print(n)       # 100, n is changed by func3
print(m)       # 100

m = func3(n)
print(n)       # 1000
print(m)       # 1000

m = func3(5)
print(n)       # 5000
print(m)       # 5000
```

Side effect

A function or expression has a side effect if it modifies some state outside its local environment.

Functions

- Global objects
  - Global objects cannot be parameters

```
def func4(n):  # SyntaxError
    global n
    return n
```

  - Global objects cannot be initialized

```
def func5(x):
    global n = 0 # SyntaxError
    n *= x
    return n
```

  - Duplicate declaration of global object is allowed

```
def func6(x):
    global n   # n is a global object
    n *= x
    return n
global n       # OK! n is a global object
n = 10
m = func6(n)
```

Modules

- We can pack many function definitions into a .py file
- A module is a .py file containing Python definitions and statements
- For example, James.py contains four functions

```
def avg(L):
    return sum(L) / len(L)

def printList(L):
    i = 0
    n = len(L)
    while i < n:
        print('[', i, ']', L[i], sep = '')
        i += 1

def sumList(L1, L2, start = 0, stop = 0):
    if stop <= start:
        stop = min(len(L1), len(L2))
    i = start
    Lr = []
    while i < stop:
        Lr.append(L1[i] + L2[i])
        i += 1
    return Lr

def swap(a, b):
    return b, a
```

Modules

- How to use a module?
- **`import`** `module_name`
- Usage
  - `module_name.function`

```
import James
L1 = [4, 5, 6, 7, 8]
L2 = [2, 3, 4 ,5 ,6]
print(James.avg(L1))
L3 = James.sumList(L1, L2)
L1, L2 = James.swap(L1, L2)
James.printList(L1)
```

Modules

- **import** module_name **as alias**

```
import James as J
L1 = [4, 5, 6, 7, 8]
L2 = [2, 3, 4 ,5 ,6]
print(J.avg(L1))
L3 = J.sumList(L1, L2)
L1, L2 = J.swap(L1, L2)
J.printList(L1)
```

Modules

- **from** module_name **import** item_name

```
From James import avg
L1 = [4, 5, 6, 7, 8]
L2 = [2, 3, 4 ,5 ,6]
print(avg(L1))                          # OK
L3 = sumList(L1, L2)                    # NameError
L1, L2 = James.swap(L1, L2)             # NameError
James.printList(L1)                     # NameError
```

Modules

- Simple importing
  - The module files and the importing files are placed in the same folder
- The module files are placed in a subfolder
  - `import subfolder.module_name`
  - For example, if James.py is placed in a subfolder named lib
    - `import lib.James`
- Importing a module with a full path
  - **`imp.load_source(module_name, path)`**
  - `imp` is the standard library of module importing

```
import imp
J = imp.load_source('James', 'C:/CloudStation/James.py')
L1 = [4, 5, 6, 7, 8]
L2 = [2, 3, 4 ,5 ,6]
print(J.avg(L1))
L3 = J.sumList(L1, L2)
L1, L2 = J.swap(L1, L2)
J.printList(L1)
```

Modules

- Let's try it
  - Design two functions to convert temperature between Fahrenheit and Celsius
    - `toC(F)`
      - Fahrenheit (ºF) to Celsius (ºC)
      - °C = (°F - 32) x 5/9
    - `toF(C)`
      - Celsius (ºF) to Fahrenheit(ºC)
      - °F = (°C × 9/5) + 32
  - Pack these functions into a module named `temperature`
  - Try to let the following program can be executed correctly

```
import temperature
print(temperature.toC(75.2))      # 24
print(temperature.toF(34.5))      # 94.1
```

Lambda functions

- A lambda function is a temporary function with a single expression
- Syntax:
  - `lambda parameter1, parameter2, …, parameterN: expression`
- For example, define a special comparison rule for sorting

```
Ls1 = ['cat', 'mouse', 'pig', 'dog', 'bird']
Ls2 = sorted(Ls1)
Ls3 = sorted(Ls1, key = lambda x: len(x))
print(Ls2)     # ['bird', 'cat', 'dog', 'mouse', 'pig']
print(Ls3)     # ['cat', 'pig', 'dog', 'bird', 'mouse']
```

Lambda functions

- Let's try it
  - Modify the following code such that a list of numeric strings can be sorted by the numeric value of each string.
  - Try to let the following program can be executed correctly

```
L1 = ['123', '000999', '54', '7.1', '   88']
L2 = sorted(L1, key =  ???  )
print(L2)      # ['7.1', '54', '   88 ', '123', '000999']
```

Exercise 1

- Design a function named `innerproduct`. It has four parameters that are
  - L1: list 1
  - L2: list2
  - start: the start index
  - stop: the stop index
- Then, `innerproduct` can compute the inner product of two lists by the following equation:

$$L_1 \cdot L_2 = \sum_{i=start}^{stop-1} L_1[i] \times L_2[i]$$

Exercise 2

- We can use `count` method to get the number of occurrences of an object in a list

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

- Design a function named `analyze` that has one parameter, which is a list of string.
- `analyze` returns four data that are
  - The most frequently occurring word
  - The number of occurrences
  - The longest word
  - The length of the longest word

map

- `map( )` function returns a map object (which is an iterator) of the results after applying **the given function** to each item of **a given iterable dataset** (list, tuple etc.)

```
def sqare(x):
    return x * x

L = [1, 2, 3, 4, 5]

M = map(sqare, L)
print(M) # ??

ML = list(M)
print(ML) # [1, 4, 8, 16, 25]
```

map

- You should covert the map object to a list, set, or another data container.
- However, the conversion only allowed once

```
s = '2 5 8 1 9 2 5 2'
M = map(int, s.split())

ML = list(M)
MS = set(M)

print(ML)
print(MS) # ??

MS = set(map(int, s.split()))
print(MS)
```

map

- With Lambda

```
Scores = {'Yamamoto':43, 'James':82, 'Mary':98, 'Bill':52, 'Tina':75}
Grades = list(
           map(
             lambda x:(x, 'P' if Scores[x] >= 69 else 'F'), Scores
           )
          )
print(Grades)

# [('Yamamoto', 'F'), ('James', 'P'), ('Mary', 'P'), ('Bill', 'F'), ('Tina', 'P')]
```

Exercise

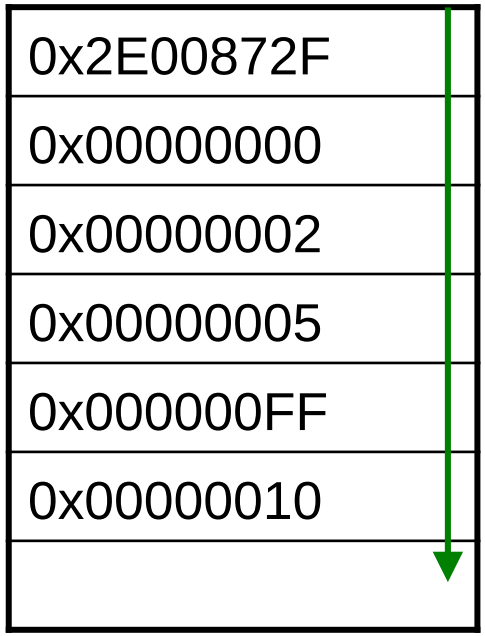
- Using map and Lambda to create a list of student IDs
  - Input: a set of integer [x1, x2, ….xn], where the maximum number is xk.
  - Output:
    `yi = 'Y00…' + str(xi)`,
    such that `len(yi) = len(str(xk)) + 1`
- You CANNOT use the loop keywords, `for` and `while`!
- For example,
  if `L = [1, 10, 12, 2, 3, 320, 506]`, then the output is
  `['Y001', 'Y010', 'Y012', 'Y002', 'Y003', 'Y320', 'Y506']`
- Another example,
  if `L = [5, 10, 12, 7, 6]`, then the output is
  `['Y05', 'Y10', 'Y12', 'Y07', 'Y06']`

Functions' calling and return

- **Call stack**

```
x = 2, y = 3
z = f(x, y)
```

```
def f(a, b):
    b = a + b
    return b
```



(1) Push the return address

(2) Reserve the return value

(3) Push the arguments

(4) Push the local variables

(5) Pop the return address

(6) Access the return value

The size of call stack depends on OS.

Recursions

- A recursion is a function that calls itself, either directly or indirectly.
- For example, summation $\sum_{k=1}^{n} k$

```
def sum(n):
    if n <= 0:
        return 0
    return n + sum(n - 1)

print(sum(10))      # 55
print(sum(20))      # 210
print(sum(2971))    # RecursionError
```

- All recursive functions can be modified to iterative functions but not vice versa.
- Only non-infinite function can be described recursively directly.

Recursions

- Example: Digits counter for an integer

```
def digitCount (x):
    if abs(x) < 10:
        return 1
    return 1 + digitCount(x / 10)


print(digitCount(10))      # 2
print(digitCount(8051))    # 4
print(digitCount(-910)     # 3
print(digitCount(0))       # 1
```

Recursions

- Example: x^y , where x is an integer and y is a positive integer

```
def powi(x, y):
    if y == 0:
        return 1
    elif y & 1:
        return x * powi(x, y - 1)
    else:
        x = powi(x, y >> 1)
        return x * x;
```

if y is even ➔ $x^y = (x^{y/2})^2$
otherwise, $x^y = x(x^{y-1})$

Loop version:

```
def powi2(x, y){
  r = 1
  while y > 0:
      while y & 1 == 0:
          x *= x
          y >>= 1
      r *= x
      y -= 1
  return r
```

Recursions

- Exercises
  - Design a recursion named **`isPalindrome`** to check whether a list is a palindrome.
    - DO NOT use `reverse` method.
  - Using **`isPalindrome`** check a text is a palindrome
    - For example:
      - **'Hello guys!'** is not a palindrome
      - **'I did did I'** is a palindrome
      - **'you are you'** is a palindrome
      - **''** is a palindrome