Python Programming

Dr. Cao Tien Dung
School of Engineering - Tan Tao University

Vicohub
https://vicohub.com

Lesson 2 & 3- Outline

- Data Structure: String, List/Tuple, Set, Dict
  - https://docs.python.org/3/tutorial/datastructures.html
- Control Flow
  - If-else
  - For
  - While
- Practice

Data type

- Some of the important types: Number, String, List, Tuple, Set, Dictionary.
  - Number: int, float and complex
  - String: is sequence of unicode characters.
  - List/Tuple is an ordered sequence of items, only difference is mutable and immutable.
  - Set is an unordered collection of unique items.
  - Dictionary is an unordered collection of key-value pairs

Operators

- Carry out arithmetic or logical computation
  - Arithmetic operators: +, -, *, /, %, //, **
  - Comparison operators: ==, !=, <, <=, >, >=
  - Logical operators: and, or, not
  - Assignment operators: =, +=, -=, *=, /=, %=, //=, **=
    - Assign to multiple names at the same time.
      - x, y, z = 2, “hello”, [1,2,3]
  - Identity operators: is, not is
    - two variables are located on the same part of the memory?
  - Membership operators: in, not in

String, List & Tuple

- String (immutable): defined using quotes (“, ‘, or “““)
  - Regular strings use 8-bit characters. Unicode strings use 2-byte characters.

    st = “Hello World” or ‘Hello World’ or “““Hello world”””
  - https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- List (mutable)
  - Ordered sequence of items of mixed types

  li = [“abc”, 34, 4.34, 23]
  - https://docs.python.org/3/library/stdtypes.html#lists
- Tuple (immutable), similarly as List but cannot **modify**

  x = 24, 3, 2018

  y = ("python", "beginner", "course")

  z = (99.99, "python", "programming", 0)

Tuple

- When and Why use Tuple rather than List:
  - Tuples are faster than List
  - When we don't have to change our data, we use Tuples
  - Tuples can be used as keys in dictionaries
  - Return multiple values in a function
- All operator as list except modify data
  - Access index and slicing
  - Mix type
  - Concatenate (+ operator)
  - Repetition (* operator)
  - Convert string or list to tuple by tuple() function
  - Unpack a tuple: `x,y = (99, 'python')`

Index & Slicing

- We can access individual members of a String, List or Tuple using square bracket "array" notation [ ]. Note that all are 0 based.
  - Positive index: count from the left, starting with 0.
  - Negative lookup: count from right, starting with –1.
- Slicing: return copy of a subset
  - Syntax
    - copy_string = string[start: end: step] # stop before end
    - copy_list = list[start: end: step] # stop before end
  - Copy whole list (also string): copy_list = list[:]
    - list2 = list1 # 2 names refer to 1 ref. change one affects both
    - list2 = list1[:] # 2 independent copies, two refs

The +, in operator

- The + operator produces a new String, List, Tuple whose value is the concatenation of its arguments.

  [1, 2, 3] + [4, 5, 6] ➔ [1, 2, 3, 4, 5, 6]

  “Hello” + “ ” + “World” ➔ ‘Hello World’

  (1, 2, 3) + (4, 5, 6) ➔ (1, 2, 3, 4, 5, 6)

- The in operator: boolean test whether a value is inside a collection

  >>> t = [1, 2, 4, 5]

  >>> 3 in t ➔ False

  >>> 4 in t ➔True

  >>> 4 not in t ➔False

  >>> s=‘Python’

  >>> ‘Py’ in s ➔True

Operations on Lists Only

>>> li = [1, 11, 3, 4, 5]

>>> li.append('a') ➔ li = [1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i') ➔ li =[1, 11, 'i', 3, 4, 5, 'a']

- + creates a fresh list (with a new memory reference)
- extend is just like add in Java; it operates on list *li* in place.

>>> li.extend([9, 8, 7]) ➔ li = [1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]

- Confusing:
  - extend takes a list as an argument unlike Java
  - append takes a singleton as an argument.

>>> li.append([10, 11, 12]) ➔ li = [1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]

Operations on Lists Only

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')      ➔ 1 # index of first occurrence*
>>> li.count('b')      ➔ 2 # number of occurrences
>>> li.remove('b')     ➔ li= ['a','c','b'] # remove first
occurrence
 ------------------
>>> li = [5, 2, 6, 8]
>>> li.reverse()       ➔ li = [8,6,2,5] # reverse the list in place
>>> li.sort()          ➔ li = [2,5,6,8] # sort the list in place
```

- list.sort(key=function) #sort complex items

```
>>> list = [(2, 2), (3, 4), (4, 1), (1, 3)]
>>> list.sort(key=lambda x: x[1])     # sort tuple using 2nd item
[(4, 1), (2, 2), (1, 3), (3, 4)]
```

Practice

- Write a python script to:
  - Get a string from console s1
  - Extract 3 middle characters of s1
  - Input second string s2
  - Insert s1 into middle of s2
- Given a list
  - l = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
  - Remove all number from the list??

If statement

- Do something If a condition is True

```
if test_expression:
    statement
```

- Do this if condition is True, otherwise do that

```
if test_expression:
    statement
else:
    statement
```

if – elif- else statements

- Sometime in our program we want to make decision based on some conditions and then perform different actions for different cases of the outcome.

```
x = int(input('Enter a choice: '))
if x == 3:
      print("X equals 3.")
elif x == 2:
      print("X equals 2.")
else:
      print("X equals something else.")
print("This is outside the 'if'.")
```

Nested if-else statements

- Sometime, you may want to make clear test expression instead of using a complex one.

```
num1 = int(input("Enter 1st number: "))
num2 = int(input("Enter 2nd number: "))
if num1 >= 0:
    if num2 >= 0:
        print("Excellence!")
    else:
        print("Fair 1!")
elif num2 >=0:
    print("Fair 2!")
else:
    print("Ignore!")
```

Conditional Expressions

- Assign value to a variable based a test condition

```
x = true_value if condition else false_value
```

  - Example:

>>> a = 2

>>> x = 1 if a == 2 else -1

out: x = 1

Why Loops?

- All programming languages need ways of doing similar things many times, this is called iteration.
- For loop
  - The for statement is used to iterate over the elements of a sequence.
  - It's traditionally used when you have a piece of code which you want to repeat n number of time.
  - The for loop is often distinguished by an explicit loop counter or loop variable.

for loop

- For-each is Python's only form of for loop

```
for <item> in <collection>:
    <statements>
```

  - collection: list, tuple, string, set, dictionary
  - item: can be more complex than single variable
  - use function range(start, end) if you want to run over a sequence. range(5) → return [0,1,2,3,4]

```
for i in range(5)
    print(i)
=====
for (x, y) in [('a',1),('b',2),('c',3),('d',4)]:
    print(x)
```

for loop (cont.)

- Don't use range() to iterate over a sequence solely to have the index and elements available at the same time

```
Avoid (still work)
for i in range(len(mylist)):
      print (“{},{}”.format(i, mylist[i]))
Use
for (i, item) in enumerate(mylist):
      print (“{},{}”.format(i, item))
```

While loop

- The while loop tells the computer to do something as long as the condition is met.
- Its construct consists of a block of code and a condition.
- The condition is evaluated, and if the condition is true, the code within the block is executed.
- This repeats until the condition becomes false.

```
a = 0
while a < 10:
    a = a + 1
    print(a)
```

While loop example

- This will ask the user for an input until he/she enter a "stop" word.

```
stop = false
while not stop:
      s = input('enter text or type stop to quit')
      if s == 'stop':
            stop = True
      else:
            print("text consists of {} words"
                  .format(len(s.split())))
```

while loop and break, continue

- You can use the keyword **break** inside a loop to leave the **while** loop entirely.
- You can use the keyword **continue** inside a loop to stop processing the current iteration of the loop and immediately go on to the next one.
- Try

```
x = 1
while x < 10:
    print(x)
    x += 1
    if x == 5:
        break
```

```
x = 1
while x < 10:
    x += 1
    if x % 2 == 0:
        continue
    print(x)
```

List comprehension

- List comprehensions provide a concise way to create lists from the iterable objects.

```
arr = range(0,11)
newList = []
for x in arr:
    newList.append(x**2)
```

```
[ expression for name in
iterable object ]
```

```
newList = [x**2 for x in arr]
```

- Filter

```
[expression for name in iterable object if test]
```

  - Get odd value from list

```
arr = range(0,11)
odd = []
for x in arr:
    if x%2 !=0:
        odd.append(x)
```

```
odd = [x for x in arr if x%2!=0]
```

List comprehension with if-else

- Generate a list with if-else expression

```
[ if_expression if test else else_expression
for name in iterable object ]
```

- Example

```
mytuple= (0,1,2,3,4,5,6,7,8)
a = [ x/2 if x %2 == 0 else x**2 for x in mytuple ]
#output
[0.0, 1, 1.0, 9, 2.0, 25, 3.0, 49, 4.0]
```

Nested loop in list comprehension

```
m = [[1,2,3],[4,5,6]]
mul = []
for x in m:
    mul_r = []
    for i in x:
        mul_r.append(i*2)
    mul.append(mul_r)
```

```
#list comprehension
mul = [[i*2 for i in x] for x in m]
```

Set

- A set contains an unordered collection of unique and immutable objects.
- If we want to create a set, we can call the built-in set function with a sequence or another iterable object:

```
x = set("A Python Tutorial")
x = set(["Perl", "Python", "Java"])
x = {1, 1, 2, 2, 3, 3} ➔ {1,2,3}
```

- Set can't contain mutable object

```
x = {1, 2, [3, 4]}  ➔ ERROR
x = {1, 2, (3, 4)} ➔ OK
```

Set

- Set is mutable object, but you can change it to immutable by **`frozenset()`** function

```
x = frozenset({1, 2, 3})
x.add(4) ➔ ERROR
```

- Set operations
  - **`add(element), clear(), copy(), difference(set), discard(element), union(set), intersection(set), issubset(set), isuperset(set)`**

Dictionaries

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type.
  - Values can be any type
  - Values and keys can be of different types
- You can
  - define
  - modify
  - view
  - lookup
  - delete

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>>d['id'] = 45
>>>d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.

Dictionary function

- d = {'user':'bozo', 'p':1234, 'i':34, 1:24}
- d.get(key) ,🡐🡒 dict[key] #return value by key
- d.pop(key) # return value by key and remove it from dictionary
- d.update(dict) # merges a dictionary with another dictionary or with an iterable of key-value pairs.
- d.clear() # Remove all.
- d.keys() # List of current keys 🡒 [‘user’, ‘p’, ‘i’, 1]

Dictionary function

- d.items() # List of item tuples. ➔ [(‘user’,‘bozo’), (‘p’, 1234), (‘i’,34), (1,24)]
- del(d['p']) ➔d = {'user': 'bozo', 'i': 34, 1: 24} # del is python function, also work for list
- Dictionary from list
  - x = [1, 2, 3]
  - y = ['a', 'b', 'c']
  - dict(zip(x, y)) ➔ {1: 'a', 2: 'b', 3: 'c'}

Output formatting

- 2 ways of writing values: **`print()`** and **`write()`** for file object. and also 2 ways to format an out.
  - handle by yourself, using string slicing and concatenation operations

```
a = 10
s = 'hello'
w = 'world'
print(s + ' ' + w, a , 'times')
```

  - use **`str.format()`**

```
print('{} {} {} times'.format(s,w,a))
x = 12.2413579
print('x = {:.2f}'.format(x))
```

Output formating

- to put a number into string, we have to convert to string using **`str()`** or **`repr()`**

```
>>> s = 'Hello World'
>>> str(s)
'Hello World'
>>> repr(s)
"'Hello World'"
>>> str(1/7)
'0.14285714285714285'
```

```
>>> print('Int value ' + repr(4.0))
Int value 4.0
>>> print(repr('hello\n'))
'hello\n'
>>> repr((30, 50.2, ('one','two')))
"(30,50.2, ('one','two'))"
```

Output formatting

- 2 ways to write a table of squares and cubes

```
for x in range(1, 11):
      print(repr(x).rjust(2), repr(x**2).rjust(3),
                                    repr(x**3).rjust(4))

for x in range(1, 11):
      print('{0:2d} {1:3d} {2:4d}'.format(x, x**2, x**3))
```

value index

space number

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

- str.rjust() – right justifies
- str.ljust(), str.center()

https://docs.python.org/3.4/library/string.html

Output formatting

- Python 3.6+ added a new string formatting approach called formatted string literals or “f-strings”. This new way of formatting strings lets you use embedded Python expressions inside string constants

```
x = 12
y = 24.369
print(f'{x} + {y} = {x + y}')
```

Practice

- https://docs.google.com/document/d/1g7EvXmXAlnWWtej2RAi_V0ijkXX0lGtRUt9bK5Jhu-U/edit?usp=sharing