2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 1-4:
More on sequence types & data structures

Kai-Feng Chen
National Taiwan University

SEQUENCE TYPES: STRINGS, LISTS AND TUPLES

- These sequence types (*string, list and tuple*) are one of the core features of python. Very important and extremely useful!
- A sequence is a container of objects which are kept in a specific order. The individual objects in a sequence can be identified by their position or index.
  - **String:** or str, a container of single-byte ASCII characters.
  - **Tuple:** a container of anything with a fixed number of elements.
  - **List:** a container of anything with a dynamic number of elements.

Tuples and strings are **immutable**. We can examine the object, looking at specific characters or items, but we cannot change the object. On the other hand, **Lists** are **mutable**.

SEQUENCE TYPES

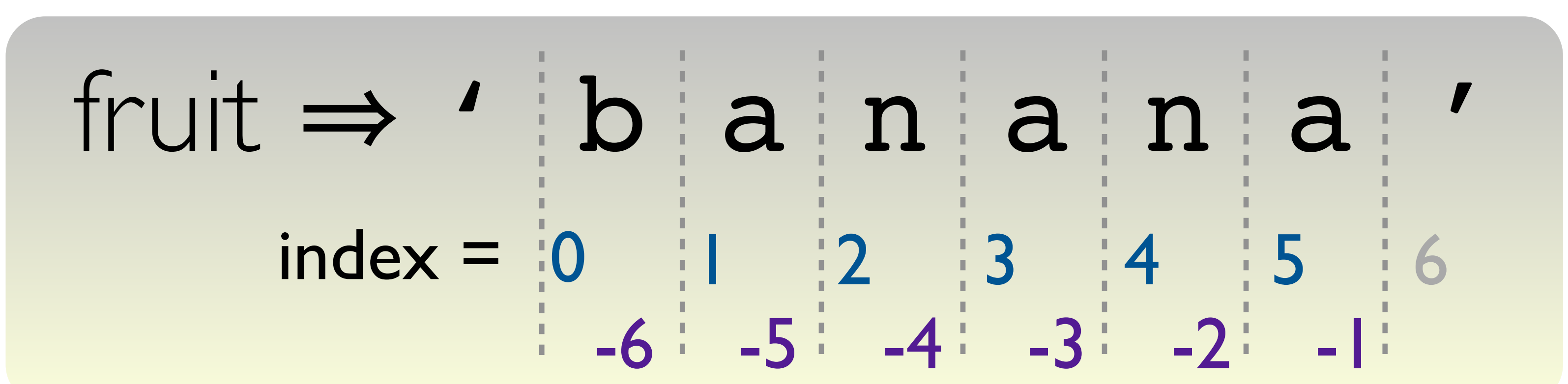
- All the sequence types have common characteristics.
- Literal values — each sequence type has a literal representation:
  - String uses **quotes** : 'string' or "string".
  - tuple uses (): (1,'b',3.1).
  - list uses []: [1,'b',3.1].
- Operations — there are three common operations:
  - + will concatenate sequences to make a longer one.
  - * is used with a number to repeat the sequence several times.
  - [ ] operator is used to select elements.

We will go through these 3 sequence types (in details).

STRINGS REVISIT

- We have slightly "touched" strings already in the previous lectures. But the python strings are much more powerful than that.
- A string contains a sequence of characters, which can be accessed with the bracket operator:

```
>>> fruit = 'banana'
>>> fruit[0] ⇐ indexing from left-hand side
'b'
>>> fruit[-1] ⇐ indexing from right-hand side
'a'
```



COUNTING AND SLICING

- **Counting:** the function **len()** returns # of characters in a string.
- **Slicing:** operator **[n:m]** returns the part of the string from the "n-th" character to the "m-th" character:
  - The first character (n) is included.
  - The last character (m) is **NOT** included.

```
>>> fruit = 'banana'
>>> fruit[1:4]
'ana'
>>> fruit[1:-2]
'ana'
>>> fruit[:3]  ⇐ start from the first character
'ban'
>>> fruit[3:]  ⇐ extend to the last character
'ana'
```

STRINGS ARE IMMUTABLE

- It is not allowed to use the [] operator on the left side of an assignment, with the intention of changing a character in a string:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- Solution: create a new string that is a variation on the original:

```
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

STRING METHODS

- Similar to functions — methods take arguments and returns a value, but with a slightly different syntax.
- Examples:

```
>>> word = 'banana'
>>> new_word = word.upper() ⇐ instead of upper(string),
>>> print(new_word)            the syntax is string.upper()
BANANA
>>> 'ORANGE'.lower()
'orange'
```

- Get the full help of string methods:

```
>>> help(str)
```

FIND METHOD

- The method find() determines if a specific character/substring occurs in string, or in a substring of string if starting index beg and ending index end are given.
- A couple of examples:

```
>>> word = 'banana'
>>> index = word.find('a') ⇐ return the index of the first
>>> print(index)             character found in the string.
1
>>> word.find('na') ⇐ can be a substring rather than a character.
2
```

FIND METHOD (II)

- The full syntax is [*just type* **`help(str.find)`** *to show it*]:

```
find(...)
    S.find(sub [,start [,end]]) -> int
```

- The start/end are the starting/ending index in the search:

```
>>> word.find('na')
2
>>> word.find('na', 3)
4
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

⇐ the ending index is not included as well.

THE IN OPERATOR

- The **find()** method should be used only if you need to know the position of a substring. To check if something is in the string or not, it is better to use the **in** operator.
- Similarly the **not in** operator works just in a similar way.
- For example:

```
>>> 'nana' in 'banana'
True
>>> 'seed' in 'banana'
False
>>> 'seed' not in 'banana'
True
```

COMPARISON OF STRINGS

- The relational operators can be applied to strings as well:

```
x == y    x > y    x >= y
x != y    x < y    x <= y
```

- These relational operations are based on the standard character-by-character comparison rules. For example:

```
>>> word = 'banana'
>>> 'banana' == word
True
>>> 'Banana' == word  ⇐ It's case-sensitive!
False
```

COMPARISON OF STRINGS (II)

- Few more string comparison examples:

```
>>> 'abc' > 'abc'
False
>>> 'abc' > 'Abc'   ⇐ 'A' has a smaller ASCII code than 'a'.
True
>>> 'abd' > 'abc'
True
>>> 'abcd' > 'abc'
True
>>> 'Abcd' > 'abc'  ⇐ It's comparing the characters one-by-one
False                  in sequence.
```

STRING FORMATTING

- One of the important string features is the ***string formatting***. This can be done through the **operator** %, which is unique to the strings and makes up for the pack of having functions from C's printf() family. For example:

```
>>> print('My name is %s and weight is %d kg!' % ('Zara',
21))
My name is Zara and weight is 21 kg!
>>> print('The value of pi is close to %.2f.' % math.pi)
The value of pi is close to 3.14.
```

This is a classical way in python and is not really different from C!
Remark: newer python (2.6 and above) introduced a new string method **format()** which can do a similar thing but more flexible operation.

STRING FORMATTING (II)

- The basic syntax:

The % operator

```
'%f' % 1.234567
```

The format symbol — The value to be inserted

```
'%s = %f' % ('pi',3.14159)
```

The values to be inserted (with **TUPLE** format)

- A couple of common format symbol:

```
%c character
%s string
%d signed integer
%x hexadecimal integer
%e exponential notation
%f floating point number
%g the shorter of %f and %e
```

STRING FORMATTING (III)

- Other extended functionality (examples):

```
>>> hbar = 1.054571726*10**-34
>>> hbar
1.054571726e-34
>>> print('%f %e %g' % (hbar,hbar,hbar))⇐ 3 way to present float point number
0.000000 1.054572e-34 1.05457e-34
>>> print('Serial: %05d' % 42)⇐ fill 0 up to 5 characters
Serial: 00042
>>> print('Serial: %5d' % 42)⇐ fill space up to 5 characters
Serial:    42
>>> print('Price: %9.2f' % 50.4625)⇐ precision limitation + fill space
Price:     50.46
>>> print('Rate: %+.2f%%' % 1.5)
Rate: +1.50%
```

INTERMISSION

- Given

```
>>> fruit = 'banana'
```

What are the following output?

```
>>> fruit[-0]
>>> fruit[len(fruit)]
>>> fruit[-10:]
>>> fruit[-10]
>>> fruit[3:3]
>>> fruit[:]
>>> fruit[3:10]
>>> fruit[10:]
```

LISTS

- A list is also a sequence of values. String contains only characters. In a list, they can be *any type*.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets [ ]:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> mix = ['spam', 2.0, 5, [10, 20]]
>>> print(cheeses, numbers, '\n', empty, mix)
['Cheddar', 'Edam', 'Gouda'] [17, 123]
[] ['spam', 2.0, 5, [10, 20]]
```

The character '\n' wraps to next line.

LISTS ARE MUTABLE

- Accessing the elements is the same as for accessing the characters of a string with the bracket operator.

```
>>> print(cheeses[0])
Cheddar
```

- Lists are mutable (unlike the strings!):

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

LIST INDICES

- List indices work the same way as string indices:
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an IndexError.
  - If an index has a negative value, it counts backward from the end of the list.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses[0] ⇐ indexing from left-hand side
'Cheddar'
>>> cheeses[-1] ⇐ indexing from right-hand side
'Gouda'
```

LIST COUNTING & SLICING

- The function **len()** also returns # of elements in a list.
- **Slicing:** operator **[n:m]** returns the part of the list from the "n-th" item to the "m-th" item:
  - The first item (n) is included.
  - The last item (m) is **NOT** included.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses[1:2]
'Edam'
>>> cheeses[1:]
['Edam', 'Gouda']
```

LIST METHODS

- Just like the strings, python also provides methods that operate on lists. For example, **append()** adds a new element to the end, **insert()** update the list by inserting the item at the position index.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> cheeses.append('Mozzarella')
>>> cheeses
['Cheddar', 'Edam', 'Gouda', 'Mozzarella']
>>> cheeses.insert(1,'Parmesan')
>>> cheeses
['Cheddar', 'Parmesan', 'Edam', 'Gouda', 'Mozzarella']
```

LIST METHODS (II)

- Adding a list to another list is possible, which is the **extend()** method:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> italian_cheeses = ['Mozzarella', 'Parmesan']
>>> cheeses.extend(italian_cheeses)
>>> cheeses
['Cheddar', 'Edam', 'Gouda', 'Mozzarella', 'Parmesan']
```

- If you use append() instead of extend() here:

```
>>> cheeses.append(italian_cheeses)
>>> cheeses
['Cheddar', 'Edam', 'Gouda', ['Mozzarella', 'Parmesan']]
```

DELETING ELEMENTS

- There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop()**:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t,'<-->',x)
['a', 'c'] <--> b
```

⇐ list is modified and returns the element that was removed.

- If you know the element you want to remove (but not the index), you can use **remove()**:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

DELETING ELEMENTS (II)

- The **del** operator also works in its own way:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

- Especially if you want to remove more than one element, you can use **del with a slice index**:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

FINDING IN LIST

- Find the location in a list by **index()** method:

```
>>> t = ['a', 'b', 'c']
>>> t.index('b')
1
```

- Counting a specific element in the list can be done by **count()**:

```
>>> t = ['b', 'a', 'n', 'a', 'n', 'a']
>>> t.count('a')
3
```

- More on list methods:

```
>>> help(list)
```

CONVERTING STRING TO LIST

- The string method **split()** returns a **list** of all the words in the string (splits on all whitespace if left unspecified):

```
>>> s = '''Learn from yesterday,
... live for today,
... hope for tomorrow.
... The important thing is to not stop questioning'''
>>> s.split('\n')
['Learn from yesterday,', 'live for today,', 'hope for
tomorrow.', 'The important thing is to not stop questioning']
>>> s.split()
['Learn', 'from', 'yesterday,', 'live', 'for', 'today,',
'hope', 'for', 'tomorrow.', 'The', 'important', 'thing', 'is',
'to', 'not', 'stop', 'questioning']
```

INTERMISSION

- Try this magical way to split and joint the strings:

```
>>> text = 'Englert-Brout-Higgs-Guralnik-Hagen-Kibble'
>>> l = text.split('-')
>>> l
['Englert', 'Brout', 'Higgs', 'Guralnik', 'Hagen', 'Kibble']
>>> '/'.join(l)
```

What do you find here? What's the easiest way to get a single spaceless string, e.g. "`EnglertBroutHiggsGuralnikHagenKibble`"?

INTERMISSION (II)

- What will happen if you try to append a list to itself?

```
>>> t = ['a', 'b', 'c']
>>> t.append(t)
```

Try to do it and see what you find.

- Instead of append, if you use the extend() method and “+” operator, what will you find?

```
>>> t = ['a', 'b', 'c']
>>> t.extend(t)
>>> t = t + t
```

FUNCTIONAL PROGRAMMING TOOLS

- There are three built-in functions that are very useful when used with lists: **filter()**, **map()**, and **reduce()**.
- The **filter(function, sequence)** returns a 'filter' object consisting of those items from the sequence for which function(item) is true:

```
>>> def is_odd(x):
...     return x % 2 == 1
...
>>> filter(is_odd, [1,2,3,4,5,6,7])
<filter object at 0x10c17deb8>
>>> list(filter(is_odd, [1,2,3,4,5,6,7]))
[1, 3, 5, 7]
```

FUNCTIONAL PROGRAMMING TOOLS (II)

- **map(function, sequence)** calls function(item) for each of the sequence's items and returns a "map" object containing of the return values. For example:

```
>>> def cube(x):
...     return x*x*x
...
>>> map(cube, [1,2,3,4,5,6,7])
<map object at 0x10c168320>
>>> list(map(cube, [1,2,3,4,5,6,7]))
[1, 8, 27, 64, 125, 216, 343]
>>> def add(x, y):
...     return x+y
...
>>> list(map(add, [1,2,3], [2,3,4]))
[3, 5, 7]
```

FUNCTIONAL PROGRAMMING TOOLS (III)

- **reduce(function, sequence)** returns a single value constructed by calling the binary function function on the first two items of the sequence, then on the result and the next item, and so on.
- This may not be as straightforward as the previous two calls, but it is indeed useful:

```
>>> from functools import reduce
>>> def add(x, y): return x+y
...
>>> reduce(add, [1,2,3,4,5,6,7])
28
```

Surely you can use loop to do exactly the same thing without a problem.

LAMBDA FUNCTION

- In the previous slides you may find that the code becomes "not-so-elegant" when introducing the short/simple functions.
- A solution to make your code even shorter with the **Lambda** functions.

```
>>> def is_odd(x):
...     return x % 2 == 1
...
>>> is_odd2 = lambda x : x % 2 == 1
>>> list(filter(is_odd, [1,2,3,4,5,6,7]))
[1, 3, 5, 7]
>>> list(filter(is_odd2, [1,2,3,4,5,6,7]))
[1, 3, 5, 7]
>>> list(filter(x:x%2==1, [1,2,3,4,5,6,7]))
[1, 3, 5, 7]
```

input argument

output return

⇐ this is the same as the "is_odd"!

OBJECTS AND VALUES

- If we execute these assignments and following statements:

```
>>> a = 'banana'
>>> b = 'banana'
```

Same content or same object?

To check whether two variables (a,b) refer to the **SAME** object, one can use the **is** operator (while the regular == operator check the contents).

```
>>> a is b      ⇐ same object
True
>>> a == b      ⇐ same content
True
```

Python creates only one 'banana' string in this example.

OBJECTS AND VALUES

- But when you create two lists, you actually get two objects:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
```

```
a⟶ [1,2,3]
b⟶ [1,2,3]
```

- In this case we would say that the two lists are equivalent, but not identical, because they are not the same object.
- **"a == b" does not imply "a is b":**

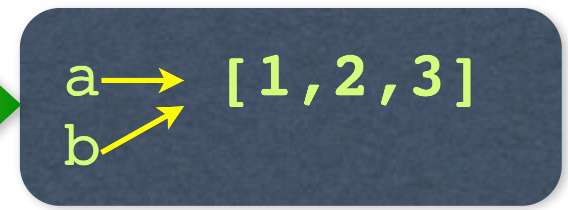
```
>>> a is b
False
>>> a == b
True
```

Python can create two separate lists with the same elements.

ALIASING

- If **a** refers to an object and you assign **b = a**, then both variables refer to the same object:

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
```



- The association of a variable with an object is called a **reference**.
- If the aliased object is mutable (such as list!), changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Be careful about this when you are developing your code!

TUPLES

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so the tuples are a lot like lists.
- The important difference is that tuples are **immutable**.
- Examples for creations of tuples:

```
>>> 'a', 'b', 'c', 'd', 'e'   ⇐ comma-separated values as a tuple
('a', 'b', 'c', 'd', 'e')
>>> ('a', 'b', 'c', 'd', 'e') ⇐ it is common to enclose
('a', 'b', 'c', 'd', 'e')        tuples in parentheses:
>>> tuple('abcde')            ⇐ The function tuple() will convert
('a', 'b', 'c', 'd', 'e')       any sequence to a tuple.
>>> ('a',) ⇐ single element tuple
('a',)
```

TUPLE (II)

- Most list operators also work on tuples. The bracket operator indexes an element as usual:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

- You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t[0] = 'A'
TypeError: 'tuple' object does not support item assignment
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

TUPLE ASSIGNMENT

- You already saw the **tuple assignment** before:

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1  ⇐ here
    while a < n:
        print (a, end=' ')
        a, b = b, a+b ⇐ here as well!
```

- It is often very useful to swap the values of two variables:

```
temp = a
a = b
b = temp
```

→

```
a, b = b, a
```

Tuple assignment is much more elegant!

TUPLES AS RETURN VALUES

- A function can only return one value, but **if the value is a tuple**, the effect is the same as returning multiple values.
- For example, the function divmod() takes two arguments and returns a tuple of two values, the quotient and remainder:

```
>>> t = divmod(7,3)
>>> t
(2, 1)
>>> quot, rem = divmod(7,3)
>>> print('quotient =',quot,'and remainder =',rem)
quotient = 2 and remainder = 1
```

When coding for your own function – you just need to do something like **"return a, b"**

DICTIONARIES

- Again a **dictionary** is similar to a list, but more general.
- Indices have to be integers in lists; in a dictionary they can be (almost) any type (which are called **keys**).
- Each key maps to a **value**.

```
>>> en2fr = {'one':'une', 'two':'deux', 'three':'trois'}
>>> en2fr['two']
'deux'
>>> en2fr['two'] = 'DEUX' ⇐ values can be modified, but not the keys
>>> en2fr['two']
'DEUX'
>>> en2fr['four'] = 'quatre' ⇐ new key-value pair can be added
>>> en2fr
{'one': 'une', 'two': 'DEUX', 'three': 'trois', 'four': 'quatre'}
```

Remark: the order of elements in dictionary may not be obvious!

DICTIONARIES (II)

- The **in** operator works on dictionaries; it tells you whether something appears as a **key** in the dictionary:

```
>>> en2fr = {'one':'une', 'two':'deux', 'three':'trois'}
>>> 'two' in en2fr
True
>>> 'deux' in en2fr
False
>>> 'deux' in en2fr.values()
True
>>> for k in en2fr:
...     print(k,'=>',en2fr[k])
...
three => trois
two => deux
one => une
```

REVERSE LOOKUP

- Lookup: given a dictionary **d** and a key **k**, it is easy to find the corresponding value **v** = d[k].
- Reverse lookup: given **d** and **v** and then find **k**. There is no simple syntax to do it, you have to search. For example:

```
>>> def reverse_lookup(d, v):
...     for k in d:
...         if d[k] == v:
...             return k
...
>>> reverse_lookup(en2fr,'trois')
'three'
```

It is obvious the performance of such search cannot be high...

INTERMISSION

- There are several methods to produce a list of n^2 like this:

```
[0,1,4,9,16,25,36,49,...,9801,9604]
```

- Try the following:
  - Write a standard loop and append the elements one-by-one.
  - Use the map() function.
  - Use the following single line list comprehensions:

```
>>> [x**2 for x in range(100)]
```

INTERMISSION

- Try to run this:

```
non = ['song','game','challenge','dream','sacrifice']
act = ['sing','play','meet','realize','offer']
for n,a in zip(non,act):
    print('Life is a %s - %s it.' % (n,a))
```

What do you see? Please also attach the missing last line "Life is love - enjoy it." to the end.

zip() function "zip" the lists to be paired items...

HANDS-ON SESSION

- **Practice 1:**
  Write a small program to print this on the screen using string format setting:

```
e =                2.72
e =               2.718
e =              2.7183
e =             2.71828
e =            2.718282
e =           2.7182818
e =          2.71828183
e =         2.718281828
e =        2.7182818285
e =       2.71828182846
e =      2.718281828459
e =     2.7182818284590
e =    2.71828182845905
e =   2.718281828459045
```

HANDS-ON SESSION

- **Practice 2:**
  Write a small program to operate on the following list:

```
[3,17,31,97,43,11,2,29,51,97,67,5,79,13,87,53,19]
```

Build a new list with the **Geometric Mean** of the two adjoint numbers, take the **floor** to integer, ie.

$$\sqrt{3 \times 17} \approx 7.1414 \quad \Rightarrow 7$$
$$\sqrt{17 \times 31} \approx 22.9565 \quad \Rightarrow 22$$
$$\ldots$$

Print the output list on the screen, and what is sum of all the numbers in the list?

```
[7,22,54,... ]
```

HANDS-ON SESSION

- **Practice 3:**
  Write a small program to count how many 0,1,2,3,4,5,6,7,8,9 in first 300 digits of π below (e.g. how many 0's, how many 1's, etc.):

```
3.141592653589793238462643383279502884197169399375105
82097494459230781640628620899862803482534211706798214
80865132823066470938446095505822317253594081284811174
50284102701938521105559644622948954930381964428810975
66593344612847564823378678316527120190914564856692346
0348610454326648213393607260249141273
```