Financial Econometric Analysis
Topic 2 - Introduction to Python Basics

Fang-Chang Kuo

National Chung Cheng University

September 19, 2020

Outline

- Python Basics
  - Types
  - Built-in Data Structures
  - Control Flow
  - In-class Problem Set

Overview

- In this lecture, we will go over the basics of Python programming.
- If you are already familiar or have some experiences with Python, you can jump right into today's in-class homework.
- Most of the materials from this lecture are coming from
  - W. Wesley. *Python for Data Analysis, 2nd Edition*. O'Reilly Media, Incorporated, 2017
- Here's another great online resource:
  - `https://quantecon.org/`

Python Structure

- **packages and modules:**
  - numpy, pandas, matplotlib
  - contain lots of useful functions or data-sets
  - import, top-level via from
- **statements:**
  - control flows: if-elif-else, for loop, while loop
  - create objects: assignments, append, slice
  - indentation matters – instead of {}
- **objects:**
  - everything is an object
  - automatically reclaimed when no longer needed
- **methods:**
  - objects may have some very useful methods.
  - for example, s.mean(),

Types

Standard Python Scalar Types

Type	Description
None	The Python **null** value.
str	String type; holds Unicode (UTF-8 encoded) strings
float	Double-precision (64-bit) floating-point number
int	Arbitrary precision signed integer
bool	A True or False value

Numeric Types

- The primary Python types for numbers are **int** and **float**. An int can store arbitrarily large numbers:

```
In [48]: ival = 17239871
In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

- Floating-point numbers are represented with the **float** type.
- Each floating-point number is a double-precision (64-bit). value.
- They can also be expressed with scientific notation:

```
In [50]: fval = 7.243
In [51]: fval2 = 6.78e-5
```

- Integer division not resulting in a whole number will always yield a floating-point number:

```
In [52]: 3 / 2
Out[52]: 1.5
```

Strings

- Python is very powerful and flexible in its built-in string processing capabilities.
- You can write string literals using either single quotes ′ or double quotes " :

```
a = 'one way of writing a string'
b = "another way"
```

- For multiline strings with line breaks, you can use triple quotes, either ′ ′ ′ or " " " :

```
c = """
This is a longer string that
spans multiple lines
"""
```

Strings 2

- Python strings are immutable; you cannot modify a string:

```
In [56]: a = 'this is a string'
In [57]: a[10] = 'f'
-------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment
In [58]: b = a.replace('string', 'longer string')
In [59]: b
Out[59]: 'this is a longer string'
In [60]: a
Out[60]: 'this is a string'
```

Strings 3

- Many Python objects can be converted to a string using the **str** function:

```
In [61]: a = 5.6
In [62]: s = str(a)
In [63]: print(s)
Out[63]: 5.6
In [64]: type(s)
Out[64]: str
```

- Strings are a sequence of Unicode characters and therefore can be treated like other sequences, such as lists and tuples (which we will explore later):

```
In [65]: s = 'python'
In [66]: list(s)
Out[66]: ['p', 'y', 't', 'h', 'o', 'n']
In [67]: s[:3]
Out[67]: 'pyt'
```

String 4 & Booleans

- Adding two strings together concatenates them and produces a new string:

```
In [71]: a = 'this is the first half '
In [72]: b = 'and this is the second half'
In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```

- The two boolean values in Python are written as **True** and **False**.
- Comparisons and other conditional expressions evaluate to either **True** or **False**. Boolean values are combined with the **and** and **or** keywords:

```
In [89]: True and True
Out[89]: True
In [90]: False or True
Out[90]: True
```

None

- **None** is the Python null value type.
- If a function does not explicitly return a value, it implicitly returns **None**:

```
In [97]: a = None
In [98]: a is None
Out[98]: True
In [99]: b = 5
In [100]: b is not None
Out[100]: True
```

Built-in Data Structures

Tuple

- **Tuple:**
  - A tuple is a **fixed-length**, **immutable sequence** of Python objects.
- The easiest way to create one is with a comma-separated sequence of values:

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)

In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))

In [5]: tup = tuple('string')

In [6]: tup
Out[6]: ('s', 't', 'r', 'i', 'n', 'g')
```

Tuple 2

- Elements can be accessed with square brackets [] as with most other sequence types.
- As in C, C++, Java, and many other languages, sequences are 0-indexed in Python (Matlab and Julia start from 1.):

```
In [8]: tup[0]
Out[8]: 's'
In [9]: tup[1]
Out[9]: 't'
In [10]: tup[5]
Out[10]: 'g'
```

- You can concatenate tuples using the + operator to produce longer tuples:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Unpacking tuples

- Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

- If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the right-hand side of the equals sign:

```
In [15]: tup = (4, 5, 6)
In [16]: a, b, c = tup
In [17]: b
Out[17]: 5
```

List

- **Lists** are **variable-length** and their contents can be modified in-place.
- You can define them using square brackets [] or using the list type function:

```
In [36]: a_list = [2, 3, 7, None]
In [37]: tup = ('foo', 'bar', 'baz')
In [38]: b_list = list(tup)
In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']
In [40]: b_list[1] = 'replacebar'
In [41]: b_list
Out[41]: ['foo', 'replacebar', 'baz']
```

Adding and removing elements

- Elements can be appended to the end of the list with the **append** method:

```
In [42]: b_list.append('dwarf')
In [43]: b_list
Out[43]: ['foo', 'replacebar', 'baz', 'dwarf']
```

- Using **insert** you can insert an element at a specific location in the list:

```
In [47]: b_list.insert(1, 'red')
In [48]: b_list
Out[48]: ['foo', 'red', 'replacebar', 'baz', 'dwarf']
```

- The inverse operation to **insert** is **pop**, which removes and returns an element at a particular index:

```
In [49]: b_list.pop(2)
Out[49]: 'replacebar'
In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

Slicing

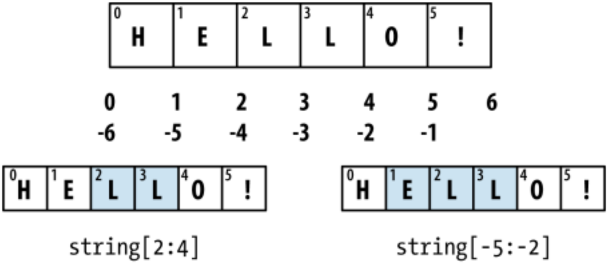
- You can select sections of most sequence types by using slice notation, which in its basic form consists of **start:stop** passed to the indexing operator **[]**:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

- Negative indices slice the sequence relative to the end:

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

Slicing and Indexing



Dict

- A **dict** is a flexibly sized collection of key-value pairs, where key and value are Python objects.

```
In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}


In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

- You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

Control Flow

IF

- Python has several built-in keywords for conditional logic, loops, and other standard control flow concepts found in other programming languages.
- The first one is **IF**.
- The **if** statement is one of the most well-known control flow statement types. It checks a condition that, if **True**, evaluates the code in the block that follows:

```
if x < 0:
    print('It's negative')

if x < 0:
 print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

FOR LOOPS

- **for loops** are for iterating over a collection (like a list or tuple) or an iterater. The standard syntax for a for loop is:

```
for value in collection:
    # do something with value
```

```
for i in range(4):
    for j in range(4):
        if j > i:
            break
        print((i, j))
```

- The **break** keyword only terminates the innermost for loop; any outer for loops will continue to run.

WHILE LOOPS

- A **while loop** specifies a condition and a block of code that is to be executed until the condition evaluates to **False** or the loop is explicitly ended with break :

```python
i = 1
while i < 6:
    print(i)
    i += 1
```

TRY EXCEPT

- Exceptions can be handled using a **try** statement.
- A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.

```
randomList = ['a', 1, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        print("The reciprocal of",entry,"is",r)
        print()
    except:
        print("Oops!")
        print("Next entry.")
        print()
```

In-class Problem Set

Problem Set 1

1. Using **str** and **int** multiplication, write a Python statement which outputs the following line:
   `HelloHelloHelloHelloHelloHelloHelloHelloHelloHello`
2. Write a Python function that takes a positive integer and returns the sum of the square of all the positive integers smaller than the specified number. Specifically, $f(4) = 3^2 + 2^2 + 1^2 = 14$
3. Define a function, $divide()$, which sequentially divides a list of numbers, and outputs the result as the final quotient. For example, $x = [10, 5, 1]$, and $divide(x) = 10 \div 5 \div 1 = 2$. (Assume the input list contains only positive integers.)
4. Write a Python function, $mean()$, which can calculate the average of the numerical items in a list and handle exceptions, non-numerical elements. For example, $a = [1, 2, `xxx`, 3, `yyy`, 4, 5]$, $mean(a) = 3.0$