Python R training course - Session 2

Van-Duyet LE, Thinh PHAM

August 12, 2017

1 Lecture list

1. Introduction to Python
2. **The Python Standard Library, if/loops statements**
    - Some useful and basic built-in functions (abs, etc.)
    - How to use import to import libraries (ex: math, random). We will introduce you several basic libraries that are often used in Python.
    - Introduce Comparison and Boolean Operators (if/else/while/for)
    - Students are also introduced the concept of object-orientation in Python.
3. Vector/matrix structures, `numpy` library
4. Python data types, File Processing, `Pandas` library
5. Functions in Python, Debugging.
6. Introduction to R, R for Python programmers.
7. Import data, plot data.
8. Data Mining in Python/R.

2 Some useful and basic built-in functions

List build-in functions: https://docs.python.org/2/library/functions.html

- **str()** is very common and useful for converting to strings, e.g. the number 1 to "1" -- similarily int() for converting other numbers (float or decimal), or strings, to integers.
- **len()** is perhaps most common, we often need to know how many members there are e.g. in a list and len(mylist) is the Python way for that (not like mylist.length in Javascript)
- **open()** is essential as that's what you use to open a file.
- **min()** and **max()** are common. also **all()** and **any()** are handy to know for basic logic.
- **set()** is useful because the set type doesn't have a literal like **[]** and **{}** for lists and dicts. **list()** is useful and quite common for converting e.g. a set to a list.
- **range()** for iterating number of times: `for i in range(10)`
- **sorted()** - creates sorted copy of list
- **help([object])** - To invoke the built-in help.

Built-in Functions				
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

2.0.1 Help

```
In [1]: help(range)
```

```
Help on built-in function range in module __builtin__:

range(...)
    range(stop) -> list of integers
    range(start, stop[, step]) -> list of integers

    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

2.0.2 Type conversion & Type coercion

Python provides a collection of built-in functions that convert values from one type to another.

```
>>> int(5.6)
    5

>>> float(6)
    6.0

>>> 'Two, ' + str(2)
    'Two, 2'

>>> bool(6)
    True
    ...
```

2.0.3 Math

```
In [2]: max(91, 100)

Out[2]: 100

In [3]: min(57, min(24, 44))

Out[3]: 24

In [4]: for i in range(5, 10):
            print i ** 2

25
36
49
64
81


In [5]: x = [1, 2, 3]
        y = ['a', 'b', 'c']

        zip(x, y)

Out[5]: [(1, 'a'), (2, 'b'), (3, 'c')]

In [6]: sorted([5,3,2,5,7,8])

Out[6]: [2, 3, 5, 5, 7, 8]
```

2.0.4 Strings

We have seen three types: `int`, `float`, and `string`. **Strings** are qualitatively different from the other two because they are made up of smaller pieces - characters.

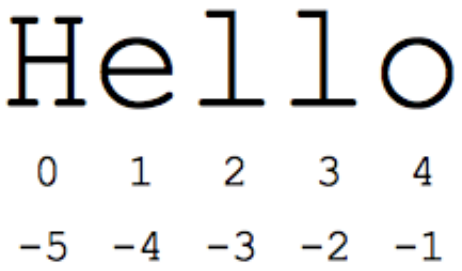
The bracket operator selects a single character from a string.

```
In [7]: fruit = "banana"
        letter = fruit[1]
        print letter

a


In [8]: len(fruit)

Out[8]: 6

In [9]: for i in range(len(fruit)):
            print 'Char at', i, 'is', fruit[i]
```

```
Char at 0 is b
Char at 1 is a
Char at 2 is n
Char at 3 is a
Char at 4 is n
Char at 5 is a
```

String slices

The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice s[start:end] is the elements beginning at start and extending up to but not including end. Suppose we have s = "Hello"




```
In [10]: s = "Hello"

In [11]: print s[0:2]

He


In [12]: print s[2:]

llo


In [13]: print s[-4:]

ello
```

3 Import libraries

The outermost statements in a Python file, or "module", do its one-time setup — those statements run from top to bottom the first time the module is imported somewhere, setting up its variables and functions.

```
In [14]: import math

         print math.sqrt(4)
2.0

In [15]: from math import sqrt

         print sqrt(4)
2.0

In [16]: from math import *

         print sqrt(4)
2.0

In [17]: import math as m

         print m.sqrt(4)
2.0
```

4 Conditionals

Ex: Input a number then assign to variable **n**. If n larger than 10, write " larger than 10", else write " doesn't larger than 10"

4.1 Boolean expressions

A boolean expression is an expression that is either true or false. One way to write a boolean expression is to use the operator ==, which compares two values and produces a boolean value:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

In the first statement, the two operands are equal, so the value of the expression is True; in the second statement, 5 is not equal to 6, so we get False. True and False are special values that are built into Python.

The == operator is one of the comparison operators; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

4.2 Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10.

`n%2 == 0` or `n%3 == 0` is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

Finally, the not operator negates a boolean expression, so `not(x > y)` is true `if (x > y)` is `false`, that is, if x is less than or equal to y.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
>>> x = 5
>>> x and 1
1
>>> y = 0
>>> y and 1
0
```

In general, this sort of thing is not considered good style. If you want to compare a value to zero, you should do it explicitly.

4.3 The "in" operator

The "in" operator could be used to check if a specified object exists within an iterable object container, such as a list:

```
In [18]: name = "John"
         name in ["John", "Rick"]

Out[18]: True

In [19]: "Duyet" in ["John", "Rick"]

Out[19]: False
```

Read python document [how to use build-in function with the argument]: https://docs.python.org/2/library/functions.html#len

4.4 IF statements

4.4.1 if-else

The basic if-else statement in Python is as follows:

```
if <expression>:
    #write statement here
else:
    #write statement here
```

```
In [20]: mark = int(input("What is your mark: "))
         if mark >= 50:
             print("Pass")
         else:
             print("Fail")
```

```
What is your mark: 10
Fail
```

4.4.2 elif

```
In [21]: month = int(input("What is the current month ? "))
         if month <=0 or month >12:
             print "Invalid month number !!!"
         elif month < 4:
             print "Spring"
         elif month < 7:
             print "Summer"
         elif month < 10:
             print "Fall"
         elif month < 13:
             print "Winter"
```

```
What is the current month ? 3
Spring
```

4.5 For loops

For loops iterate over a given sequence. Here is an example:

```
In [22]: primes = [2, 3, 5, 7]
         for prime in primes:
             print(prime)
```

```
2
3
5
7
```

For loops can iterate over a sequence of numbers using the "range" and "xrange" functions.

```
In [23]: r = range(5)
         print r
```

```
[0, 1, 2, 3, 4]
```

```
In [24]: for i in r:
             print i

0
1
2
3
4


In [25]: for i in r:
             print "I love you"

I love you
I love you
I love you
I love you
I love you
```

The **range()** function is one of Python's built in functions. It is is used to indicate how many times the loop will be repeated.

The structure of the range function is **range(start, upto, step)** in which the arguments of range are used as follows:

- **start** and step are both optional.
- **upto** must always be there, it means "up to but not including" the value.
- **start**, **upto**, and **step** must all be integers

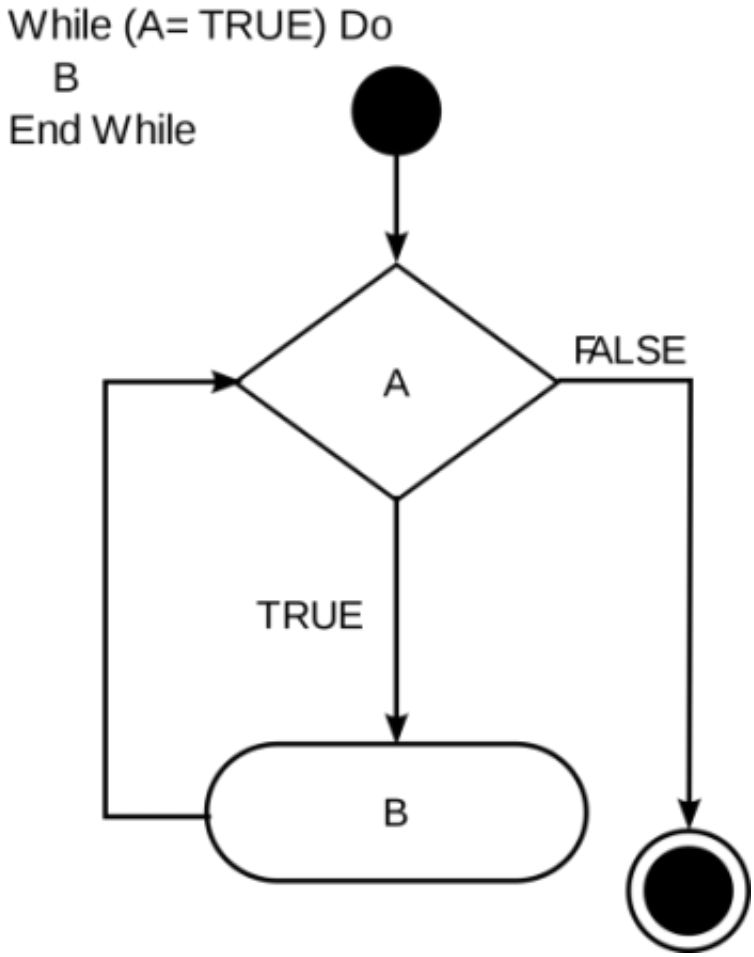
```
In [26]: range(3)

Out[26]: [0, 1, 2]

In [27]: range(3,7)

Out[27]: [3, 4, 5, 6]

In [28]: range(1,10, 2)

Out[28]: [1, 3, 5, 7, 9]

In [29]: range(10, -1, -1)

Out[29]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

4.6 While loops

While loops are known as indefinite or conditional loops. They will keep iterating until certain conditions are met. There is no guarantee ahead of time regarding how many times the loop will iterate.

The while loop, like the if statement, includes a boolean expression that evaluates to true or false. The code inside the loop will be repeatedly executed until the boolean expression is no longer true.

This diagram shows the flow of control in a while loop:




```
In [30]: count = 0
         while count < 5:
             print(count)
             count += 1  # This is the same as count = count + 1
```

```
0
1
2
3
4
```

```
In [31]: input_text = "run"
         while input_text != "exit":
             input_text = raw_input("Please type 'exit' to quit this program: ")
         print "Quit !!!"

Please type 'exit' to quit this program: hi
Please type 'exit' to quit this program: hello
Please type 'exit' to quit this program: exit
Quit !!!
```

4.7 "break" and "continue" statements

break is used to **exit** a for loop or a while loop, whereas **continue** is used to skip the current block, and return to the "for" or "while" statement. A few examples:

```
In [32]: count = 0
         while True:
             print(count)
             count += 1
             if count >= 5:
                 break

0
1
2
3
4
```

```
In [33]: for x in range(10):
             # Check if x is even
             if x % 2 == 0:
                 continue
             print(x)

1
3
5
7
9
```

4.8 Exercise

Loop through and print out all even numbers from the numbers list in the same order they are received. Don't print any numbers that come after 237 in the sequence.

Hint: Using `loop`, `continue` and `break`

```
numbers = [
    951, 402, 984, 651, 360, 69, 408, 319, 601, 485, 980, 507, 725, 547, 544,
    615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984, 592, 236, 105, 942, 941,
    386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345,
    399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687, 217,
    815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742, 717,
    958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126, 721, 328, 753, 470,
    743, 527
]

# your code goes here
```

5 References

- Class and Object: https://www.learnpython.org/en/Classes_and_Objects
- Learn Python - Loop: https://www.learnpython.org/en/Loops
- http://www.afterhoursprogramming.com/tutorial/Python/If-Statement/
- http://www.afterhoursprogramming.com/tutorial/Python/Classes/