Advanced Python

Prof. Gheith Abandah

Reference

- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
  - Material: https://github.com/wesm/pydata-book

Outline

3.1 Data Structures and Sequences

3.2 Functions

3.3 Files and the Operating System

Outline

- Tuple
- List
- Sequence Functions
- Dict

Tuple

- A tuple is a fixed-length, **immutable sequence** of Python objects.
- How to **convert objects to tuples**?
- What are the functions of operators **+** and ***** on tuples?
- **Swapping**: What is Python way to swap two vars?

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)

In [6]: tup = tuple('string')
In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')

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

In [24]: b, a = a, b
```

List

- List are **ordered** and **mutable**.

- How to **convert objects to lists**?

- What is the difference between **.append()** and **.insert()**?

```
In [37]: tup = ('foo', 'bar')
In [38]: b_list = list(tup)
In [39]: b_list
Out[39]: ['foo', 'bar']


In [44]: list(range(0, 7))
Out[44]: [0, 1, 2, 3, 4, 5, 6]


In [45]: b_list.append('baz')
In [47]: b_list.insert(1, 'red')
In [48]: b_list
Out[48]: ['foo', 'red', 'bar', 'baz']
```

List – Add and remove

- What is the difference between **.pop()** and **.remove()**?

- What is the difference between **.append()** and **.extend()**?

```
In [48]: b_list
Out[48]: ['foo', 'red', 'bar', 'baz']

In [49]: b_list.pop(0)
Out[49]: 'foo'


In [53]: b_list.remove('red')
In [54]: b_list
Out[54]: ['bar', 'baz']

In [59]: b_list.extend([7, 8])
In [60]: b_list
Out[60]: ['bar', 'baz', 7, 8]
```

7

List – Sort

- You can sort lists using **.sort()** and **sorted()**. What is the difference?

```
In [61]: a = [7, 2, 5, 1, 3]
In [62]: sorted(a)
Out[62]: [1, 2, 3, 5, 7]
In [63]: a
Out[63]: [7, 2, 5, 1, 3]


In [64]: a.sort()
In [65]: a
Out[65]: [1, 2, 3, 5, 7]
```

Syntax:

List.sort(reverse=True|False, key=myFunc)

Built-in Sequence Functions

- What does each of the following functions do?
- **enumerate()**
- **zip()**
- **reversed()**
  - What is the difference between **sorted(reverse=True)** and **reversed()**?

```python
l1 = ['foo', 'bar', 'baz']
l2 = ['one', 'two', 'three']
for i, (a, b) in enumerate(zip(l1, l2)):
    print('{0}: {1}, {2}'.format(i, a, b))
0: foo, one
1: bar, two
2: baz, three


list(reversed(range(10)))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Dictionary

- **Hash map** or **associative array** between **key-value** pairs.

- What is the difference between **.pop()** and **del**?

```python
d1 = {'a' : 1, 'b' : 2}
d1['c'] = 's'
d1
{'a' : 1, 'b' : 2, 'c' : 's'}


del d1['a']
ret = d1.pop('c')
ret
's'
d1
{'b' : 2}
```

Dictionary Methods

- Creating a dictionary

- `.update()`
- `.get()`
- `.keys()`
- `.values()`
- `.items()`

```python
mapping = {}
for key, value in zip(k_l, v_l):
    mapping[key] = value


d1.update({'b' : 11, 'c' : 12})


value = d.get(key, default_value)

for key, value in d.items():
    mapping[key] = value
```

Outline

- Namespaces, Scope, and Local Functions
- Returning Multiple Values
- Functions Are Objects
- Anonymous (Lambda) Functions
- Currying: Partial Argument Application
- Generators

Namespaces, Scope, and Local Functions

- Functions can access variables in two different scopes: **global** and **local**.
- Variables that are **assigned within** a function by default are assigned to the **local namespace**.
- What happens to **a** after the calling **func**()?

```python
def func():
    a = []
    a.append(1)
            ### ### ###
a = []
def func():
    a.append(1)
            ### ### ###
def func():
    global a
    a = []
    a.append(1)
```

Returning Multiple Values

- How to return **multiple values** from a function?

- What do you think about this **alternative**?

```python
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
    ### ### ###
def f():
    a = 5
    b = 6
    return {'a' : a, 'b' : b}
```

Functions Are Objects

- Since Python functions are objects, you can:
  - **Put them** in lists
  - **Iterate** on them

  - Use them as **arguments to other functions**

```python
string = '...'
func_list = [f1, f2, f3]
for func in func_list:
        string = func(string)


for x in map(f1, iter):
        print(x)
```

Anonymous (Lambda) Functions

- Writing functions consisting of a **single statement**

- How to **sort** a collection of strings by the **number of distinct letters** in each string?

```python
def short_function(x):
    return x * 2


equiv_anon = lambda x: x * 2


strings.sort(key=lambda x:
    len(set(list(x))))
```

```python
strings = ['card', 'aaaa', 'abab']

['aaaa', 'abab', 'card']
```

Currying: Partial Argument Application

- Currying is deriving new functions from existing ones.
- In Python, use **partial()**

```python
def add_numbers(x, y):
    return x + y

from functools import partial
add_five = partial(add_numbers, 5)
add_five(4)
9
```

Generators

- Use **iter()** to create an iterable object.

```python
some_dict = {'a': 1, 'b': 2, 'c': 3}

dict_iterator = iter(some_dict)

next(dict_iterator)
'a'

list(dict_iterator)
['b', 'c']
```

itertools module

Table 3-2. Some useful itertools functions

Function	Description
combinations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function combinations_with_replacement)
permutations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, respecting order
groupby(iterable[, keyfunc])	Generates (key, sub-iterator) for each unique key
product(*iterables, repeat=1)	Generates the Cartesian product of the input iterables as tuples, similar to a nested for loop

```
list(itertools.combinations(['a','b','c'], 2))
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

itertools module

Table 3-2. Some useful itertools functions

Function	Description
combinations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function combinations_with_replacement)
permutations(iterable, k)	Generates a sequence of all possible k-tuples of elements in the iterable, respecting order
groupby(iterable[, keyfunc])	Generates (key, sub-iterator) for each unique key
product(*iterables, repeat=1)	Generates the Cartesian product of the input iterables as tuples, similar to a nested for loop

```python
list(itertools.permutations(['a','b','c'], 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]
```

`itertools` module

- **Group** the following list by the **first letter**.

```python
names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
import itertools
first_letter = lambda x: x[0]
names.sort(key=first_letter)
for letter, n in itertools.groupby(names, first_letter):
    print(letter, list(n)) # n is a generator
A ['Alan', 'Adam', 'Albert']
S ['Steven']
W ['Wes', 'Will']
```

`itertools` module

Table 3-2. Some useful itertools functions

Function	Description
`combinations(iterable, k)`	Generates a sequence of all possible k-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function `combinations_with_replacement`)
`permutations(iterable, k)`	Generates a sequence of all possible k-tuples of elements in the iterable, respecting order
`groupby(iterable[, keyfunc])`	Generates (`key, sub-iterator`) for each unique key
`product(*iterables, repeat=1)`	Generates the Cartesian product of the input iterables as tuples, similar to a nested `for` loop

```
list(itertools.product([1, 2], ['a', 'b']))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

Outline

Python File Support

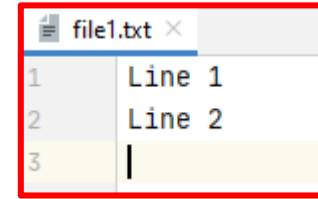- The built-in **open** function supports opening a file for reading or writing.

- You can **iterate** on the file handle.

- **Alternative syntax**

```python
path = 'folder\\file1.txt' # or /
f = open(path)  # read is default mode
for line in f:
    # remove right white space
    line = line.rstrip()
    print(line)
f.close()
```

```
Line 1
Line 2
```

```python
with open(path) as f:
    for line in f:
        print(line.rstrip())
```
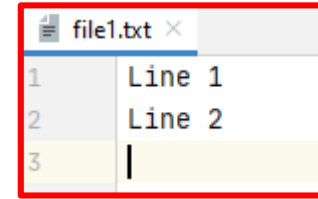
Python File Modes

Mode	Description
r	Read-only mode
w	Write-only mode; creates a new file (erasing the data for any file with the same name)
x	Write-only mode; creates a new file, but fails if the file path already exists
a	Append to existing file (create the file if it does not already exist)
r+	Read and write
b	Add to mode for binary files (i.e., 'rb' or 'wb')
t	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add t to other modes to use this (i.e., 'rt' or 'xt')

The default is text mode and utf-8 encoding.

Python File Support

- How to read a file into a **list of strings**?

- Use **read**, **tell**, and **seek** to control the reading process.

```python
with open(path) as f:
    lines = [x.rstrip() for x in f]
```

```python
>>> print(f.read(3))
Lin
>>> f.tell()
3
>>> f.seek(8)
8
>>> print(f.read(6))
Line 2
```

Important Python File Methods or Attributes

Method	Description
read([size])	Return data from file as a string, with optional size argument indicating the number of bytes to read
readlines([size])	Return list of lines in the file, with optional size argument
write(str)	Write passed string to file
writelines(strings)	Write passed sequence of strings to the file
close()	Close the handle
flush()	Flush the internal I/O buffer to disk
seek(pos)	Move to indicated file position (integer)
tell()	Return current file position as integer
closed	True if the file is closed

Writing to Files

- How to copy a text file skipping empty lines?

```python
with open('tmp.txt', 'w') as handle:
    handle.writelines(x for x in open(path) if len(x) > 1)
```

Homework

- Solve the homework on **Advanced Python Programming**

Summary

3.1 Data Structures and Sequences

3.2 Functions

3.3 Files and the Operating System