LEARNING PYTHON, PART 1: BASICS

1. Basic Operations

- Power: `a ** b`
- Mod: `a % b`
- Assignment: `=`
- Check equality: `==`
- Combine statements: use the keywords `and` for and; `or`for or. Note: `&` and `|` may also be used, but sometimes weird things happen. So `and` and `or` seem safer…
- Comment: `#`
- Stings: either `''` or `""` are fine.
- Dot `.` is NOT a usual character.
- Semi-colon `;` at the end of the sentence suppresses showing outputs.
- Boolean values: `True` and `False`.
- Use *indent (white spaces)* instead of brackets to separate a block of code.
- `None` is the Python null. To check if some object is `None`, we use `if x is None:`
- `Pass` is a place holder for an indent block.

Types of numeric values and conversions

- Usually, when data type is mixed in an operation, operands are pushed up the numeric tower:
  int -> float -> complex

- We can use the `type()` function to check the type of an object.
- **In Python 2, `int / int` is integer division, not usual division!!!**
  If we want usual division, either
  - (1) make one of the number float, e.g., `float(a) / b`, or
  - (2) add this before the division, which use the `/` in Python 3:

```
from __future__ import division
```

- In Python 3,

- `/` is regular division
- `//` is integer division

`print` function

- Note the syntax difference:

```
x = 'Hello world'
## Only in Python 2
print x
## In Python 3 (also works in Python 2)
print(x)
```

- Formatted printing using the `format` method, for example,

```
print("Pi is approximately {0:.3f}".format(22./7.))
```

Note: Each specification has two parts: `{index:format}`.
- (1) `index` starts from 0; it is the index from the list, which is the input of the `format` method.
- (2) `format`: an optional length and a mandatory one-letter data type code, such as `s` (string), `d` (integer), `f` (float). The optional length is `a.b` or `.b` or `a`, where `a` is the overall length (including sign and decimal dot), and `b` is number of decimal places.
- (3) Another example:

```
x = value1
my_str = str1
print("My {} is {}".format(my_str, x))
## The output is My str1 is value1
```

- Use `+` to cancatnate

```
print('Hello' + 'world')
```

2. List

Create a list

- Use bracket to define a list

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

- A string is a list of characters.

```
s = 'abcdefg'
```

List indexing

- *Indexing in Python starts at 0.*

```
s[0] ## returns 'a'
```

- Negative index: from the right-hand side of the string

```
s[-1] ## returns 'g'
```

- The colon `:` operator: returns the `a`th to the `(b-1)`th entries of the list:

```
s[a:b]
```

More examples:

```
s[2: ]  ## returns 'cdefg'
s[ :4]  ## returns 'abcd'
s[2:4]  ## returns 'cd'
s[start:stop:step]
s[::-1] ## returns 'gfedcba'
```

List functions

- `range` function

Returns a sequence of integers, starting from `a`, ends before but NOT including `b`. This function is often used with the `for` loop.

```
range(a, b, step = 1)
```

Note 1: In Python 3, if we want the output to be a list, we can use `list(range(a,b))`. In python 2, just use `range(a,b)`.

Note 2: `range(n)` return a sequence from `0` to `(n-1)`.

- To check if a value `val` is in a list `lst`; returns a boolean value.
  `val in lst`

- String conversion to numeric: `float('123.4')`

- Length of a list: `len(s)`

- Compute sum of a numeric list: `sum(s)`

List copies

- Suppose `x` is a list. And we define `y=x`. Then any future changes on `y` will also apply to `x`.
  Note: my understanding is that list is similar to a pointer (like in C).

- This is also the case for sets and dictionaries.
- How to copy a list, such that changing the copy doesn't affect the original list?

  - In Python 2, copy by slice
    `y = x[:]`

- In Python 3, use the `.copy` method
  `y = x.copy()`

3. Dictionary

A list with no order among components.

- Define a dictionary

```
a = {'key1': value, 'key2': value, …}
```

Or if we have a list `key` and a list `value`, then we can define the dictionary using

```
dict(zip(key, value))
```

- Visit a dictionary component

```
a['key1']  ## Note: a[0] will return an error.
```

- Add an element

```
a['key3'] = 4
```

- Delete an element

```
del a['key1']
```

- Length of a dictionary:

```
len(a)
```

4. Tuple and Set

Tuple: use `()` to define.

- A tuple cannot be edited after defined.
- *A tuple is faster to use than a list.*

```
b = (1, 2, 3)
b[0] ## returns 1
b[0] = 5 ## returns error
```

- Tuples are immutable; there are no `sort()`, `append()`, `reverse()`, etc, for tuples. This is why tuples are and faster (than lists). In this sense, tuples are somehow like strings.

- Change a dictionary `a` to a list of tuples, where each tuple is the `(key, value)` pair.

```
a.items()
```

Set: use `{}` to define.

A set only contains unique elements.

```
d = {1, 2, 2, 3} ## Actually d = {1, 2, 3}
```

Or use the `set` function to return unique elements in a list

```
## Create a set from a list
set([1, 2, 2, 3]) ## returns {1, 2, 3}

## Create a set from a dictionary
set(a) ## returns a set of the dictionary keys
```

- Set operations: suppose `d1` and `d2` are two sets.
    - `|` union
    - `&` intersection
    - `-` difference (in `d1` but not in `d2`)
    - `^` symmetric difference (in `d1` but not in `d2`, or in `d2` but not in `d1`)

- Set operation among multiple sets

```
set_name_list = [d1, d2, ...] ## is a list names of sets
set.interaction(*set_name_list)
```

- Change a set to a list

```
list(d1)
```

5. If, Elif, and Else

If function

- Use *indent (white spaces)* instead of brackets to indicate the block of code to run if the statement is `True`.
- Use colon `:` after the statement.

```
if statement:
    run1
    run2
```

```
if statement1:
    run1
elif statment2:
    run2
else:
    run3
```

6. Loops

For loops

```
for i in listx:
    run1
    run2
```

Note: when look through a dictionary or set, the order is random, because these objects are unordered.

While loops

```
while statement1:
  run1
  run2
```

Partial loops

- `break` exits from a loop
- `continue` skip over the rest of indented block (for once)

List comprehension

Applies the operation to each element of the list

```
[operation_containing_i for i in listx]
## for example
[2*i+1 for i in range(10)]
```

- We can also add an optional `if`

```
[2*i+1 for i in range(10) if i % 2 == 0]
```

- We can also use comprehension to create a set or a dictionary

```
{2*i+1 for i in range(10)} ## a set
{i: 2*i+1 for i in range(10)} ## a dictionary
```

7. Defining New Functions

Use `def` to define a new function.

```
def my_func(input1 = default1, input2 = default2):
    run1
    run2
    return x
```

- A function name starts with a lower case letter.
- Documentation string: multiple lines of comments, between two lines of three double (or single) quotes in each line. The content is what `help()` displays.
- We can return multiple things, separated by `,`. Then the returned values are in a tuple.

Vector operation: `map` function

The `map` function applies a function to each element of a list. If we want the output to be a list, then we can use the `list` function outside of the `map` function.

```
list(map(func, listx))
```

Vector operation: `filter` function

Returns the elements values in a list that satisfy the condition.

```
list(filter(condition, listx))
```

Anonymous function: `lambda` expression

For each input of `x`, this function returns the output `operation_x`.

```
lambda x:operation_x
## for example
weird_math_lambda = lambda x,y: x*y+x/y-x**2
```

A `lambda` expression is usually combined with the `map` function.

```
list(map(lambda x:operation_x, listx))
```

8. Methods

Using `object.method()` to apply a certain method on an object.

String methods

```
my_str.lower() ## change all characters to lower case
my_str.upper() ## change all characters to upper case
my_str.split(delimiter = ' ') ## split the string (by spaces)
my_str.replace('old', 'new') ## replace old by new
```

Dictionary methods

```
d.keys() ## returns the keys of the dictionary
d.items() ## returns all items, as a list of two-element tuples
d.values() ## returns the values, not necessary in any order
```

List methods

```
lst.pop([index]) ## return the last (by default) element of lst,
and meanwhile remove it from lst

lst.remove(value) ## remove the first value in the list
del lst[index] ## remove the element of this index

lst.append(value) ## append the new value to the end of lst
lst.reverse() ## reverse the list

lst.sort() ## sort in ascending order

my_str.index(value) ## Find the index corresponds to the first
fvalue
```

9. Input (stdin)

- In Python 2, use `raw_input` function.

```
raw_input('Input your name: ')
```

Alternatively, we can overwrite it to `input` (the original `input` function is Python 2 should not be used)

```
input = raw_input
input('Input your name: ')
```

- In Python 3, use the `input` function directly.

10. Reading and Writing Files

Reading (or writing) a text file

First, we need to create an `open` object

```
file_x = open('filename', mode)
```

Notes:

- `mode` is optional; it is a string containing one or more of the following:
  - `r` for reading
  - `w` for writing, which will remove previous contents in the file.
  - `a` for append.

Then, we can use the `.read` method to read:

```
file_x.read(size) ## if size is ommitted, read the whole file
file_x.readline() ## find `\n` to read a line
```

Alternative, we can also write the file

```
file_x.write('New contents')
```

Lastly, remember to close the file

```
file_x.close()
```

Reading a file using `with` (recommended!)

We don't need to worry about closing a file at the end, because the `with` statement closes the file automatically.

```
with open('file_name') as file_x:
    x = file_x.read() ## block of codes
```

Reading or writing a CSV file

Use the `csv` library. To read a csv file:

```
## Read a csv file
import csv
with open('some.csv','rb') as source_x:
    reader_x = csv.reader(source_x)
    for row in reader_x:
        print(row)
```

Notes:

- In the `open` function, the mode `b` stands for binary. In Python 2, csv files are treated as binaries.
- The `reader_x` object is an iterable. Each row is a list of column values.
- All column values are strings. If needed, number should be manually converted using `int()` or `float()`.
- We can add an optional argument in the `csv.reader` function to specify delimiter: `delimiter = '|'`

If the csv file has headings (column names), we can use

```
reader_x = csv.DictReader(source_x)
```

In this case, each row of `reader_x` is a dictionary with column names being keys.

To write a csv file:

```
## Write a csv file
import csv
with open('some.csv','wb') as target_x:
    writer_x = csv.writer(target_x)
    for row in some_source_of_data:
        writer_x.writerow(row)
```

Notes:

- `some_source_of_data` is a list (or tuple) of each row
- If we want headers, we can just write them as an extra row.

Reading JSON files

Use the `json` library.

```
import json
with open("some_file.json") as source:
    object = json.load(source)
```