Basic Python Programming

Yilin Zheng

Dept. of Computer Science and Engineering

July 10, 2017

Outline I

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Outline II

- OS

Input

- Use `input` and `raw_input`, XXXX is prompt
  - `input(XXXX)` (Python2 and 3)
    - In Python2, it only **reads numbers** and **return numbers**
    - In Python3, it reads both **numbers** and **strings** and **return only in string format**
    - `int` or `float` to get matching format
  - `raw_input(XXXX)` (Python2)
    - Reads both **numbers** and **strings** and **return only in string format**
    - `int` or `float` to get matching format

Multiple data input: eval

- Use `eval` to input multiple data in a time
- The data input should be separated by **comma**

Output

- Use `print` to output
- Formatted output: format specifier
  - %d — decimal
  - %x — hexadecimal
  - %o — octal
  - %f — float
  - %s — string
  - %c — characters
  - More on ▸ Format Specification
- Use **format specifiers** to output data in corresponding format
- Use `%(Variables)` to place the value into the result

Formatted output I: %

- The whole %: %[name][flag][width][.][precision]type
  - **name** can be empty, numbers(occupy), key of dict
  - **flag** is format qualifier:
    - +(right alignment)
    - -(left alignment)
    - #(pad 0 for oct, pad 0x for hex)
    - **0**(zero)(pad 0)
  - **width** is to control the max length
  - **precision** is to control the decimal
  - **type** is to indicate the type of the data

Formatted output II: format

- The whole `format`:
  {[name][:][[fill]align][sign][][0][width][,][.precision][type]}
  - Need to use **braces** to pass name such as **keys of dict**
  - **fill** can use any characters
  - **align** is format qualifier:
    - >(right alignment)
    - <(left alignment)
    - ^(center alignment)
    - =(length)
  - **sign** means positive(+) or negative(-)
  - Rests are the same with % format

File

- Use built-in function `open` to **open a file** and **create an object** of class **file**
- `open` need two parameters(**strings**): **file path** and **mode**
- Different **modes**
  - `'r'` — **read** mode, which is default mode if omitted; read files only
  - `'w'` — **write** mode; edit and write new information to the file (**erase any existing files with the same name**)
  - `'a'` — **appending** mode; add new data to the end of the file
  - `'r+'` — **read&write** mode; handle both read and wirte actions
  - More on ▸ Modes

Operations on files

- Common operations on files: `read`, readline, `readlines`, `write`, `writelines`, `seek`, `close`
  - **read** can return the content in the file in **string** format
  - **readline** reads **one line each time**
  - **readlines** can read **whole file and analyse to lines**
  - **write** can write **strings** to file
  - **writelines** can write multiple lines to file but need to **add newline(`n`) manually**
  - **seek** can **move pointer to certain position** of file
  - **close** is to **close files**

pickle

- Use module **pickle** to **store** and **load objects** in files, which is called **storing the object persistently**
- Use `pick.dump` to store data:
  `pickle.dump(obj, file, [,protocol])`
  - **obj** is the objects to store in
  - **file** is the storage location, a file object
  - **protocol** is an integer of protocol version
    - **0**: the **original ASCII protocol** and is backwards compatible with earlier versions of Python
    - **1**: the **old binary format** which is also compatible with earlier versions of Python
    - **2**: introduced in Python 2.3, it provides much more efficient pickling of new-style classes

pickle

- **3**: added in Python 3.0, it has explicit support for bytes objects and cannot be unpickled by Python 2.x. This is **the default protocol**, and the recommended protocol when compatibility with other Python 3 versions is required
- **4**: added in Python 3.4. It adds support for **very large objects**, pickling more kinds of objects, and some data format optimizations

- Use `pickle.load` to get objects: `pickle.load(file)`

Syntax error

- If our programs have some invalid statements, Python will prompt error
- Syntax errors, also known as parsing errors, are perhaps the most common error we will make

```
>>> def print_hello()
  File "<stdin>", line 1
    def print_hello()
                    ^
SyntaxError: invalid syntax
>>> 2\3
File "<stdin>", line 1
    2\3
      ^
SyntaxError: unexpected character after line
                                continuation
                                character
```

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when attempting to execute it. Errors detected during execution are called **exceptions** and are **not unconditionally fatal**. We can solve exceptions through error messages
- Some common exceptions:
  - Indentation error
  - Type error: e.g. interprete data to incompatible types
  - Name error: e.g. spell variables or functions wrongly
  - Index error: e.g. index out of range
  - Attribute error: e.g. wrong names of methods
  - Key error: e.g. key inexistence
  - Value error: e.g. wrong value passed to function
  - Overflow error: e.g. number is too large
  - Zero division error: use zero as denominator

Example I

```
>>> a==10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> list_1 = [1,2,3]
>>> list_1[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> def print_x(x):
...   print(x)
  File "<stdin>", line 2
    print(x)
        ^
IndentationError: expected an indented block
```

Example II

```
>>> int('hahah')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
                          base 10: 'hahah'
>>> dict_1 = {'a':1, 'b':2}
>>> dict_1['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> import math
>>> 1-math.exp(-4*1000000*-0.064151594108)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: math range error
```

try...except...

- We can use `try...except...` to handle exceptions
- We can put statements within **try block** while put error handlers within **except block**

```
#!/usr/bin/python # Filename: try_except.py
try:
    text = input('Enter something: ')
except EOFError: #set exception type
    print('Oh! You killed me!')
except KeyboardInterrupt:
    print('You cancelled the operation.')
else:
    print('You entered {}'.format(text))
```

raise

- Sometimes we need to define our own exceptions for some reason
- Use `raise` to raise an exception
- The error or exception that we can raise should be a subclass derived from the **Exception** class

```
#!/usr/bin/python # Filename: raise_exception.
                                    py
class TooShortInput(Exception): #inherit from
                                    Exception
    '''A user-defined exception class.'''
    def __init__(self, length, least):
        Exception.__init__(self)
        self.length = length
        self.least = least
```

raise

```
#continue the example
try:
text = input('Enter something: ')
if len(text) < 5:
    raise TooShortInput(len(text), 5)
except EOFError:
    print('Dead!')
except TooShortInput as ex:
    print(('TooShortInput: The input was {:d}
                           long, expected
                           at least {:d}')
                           .format(ex.
                           length, ex.
                           least))
else:
    print('No exception')
```

try...finally...

- A `finally` clause anoter choice with `try`
- `finally` is **always executed before leaving the try statement**, whether an exception has occurred or not

```
#!/usr/bin/python # Filename: try_finally.py
x,y = eval(input("Enter two integers: "))
try:
    result = x / y
except ZeroDivisionError:
    print("Cannot divided by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")
```

with...as...

- To prevent files left open or do other clean-up actions, we can use `with...as...` to allow objects like files to be used in a way that ensures they are always cleaned up promptly and correctly
- Form: `with EXPRESSION as VARIABLE:`

```
#!/usr/bin/python # Filename: with.py
with open("file.txt") as file_1:
    for line in file_1:
        print(line, end="")
```

Standard library

- The **Python Standard Library** contains many useful modules and is part of every standard Python installation
- It is important to become familiar with the Python Standard Library since many problems can be solved quickly if we are familiar with them

sys

- We ever mentioned `sys` before
- Some functions and variables in `sys`:
  - `argv`: The list of command line arguments passed to a Python script
  - `exit([arg])`: Exit current program with appointed value or messages
  - `modules`: A dictionary that maps module names to modules which have already been loaded
  - `path`: A list of strings that specifies the search path for modules.
  - `platform`: This string contains a platform identifier
  - `stdin`: Used for all interactive input
  - `stdout`: Used for the output
  - `stderr`: For interpreters own prompts and its error messages

More detail: ▸ sys for 3.6.2rc2

Example

- Try to run the script and observe the difference

```
>>> python3 use_sys.py #no arguments
>>> python3 use_sys.py 1 2 3 4 #four arguments

#!/usr/bin/python # Filename: use_sys.py
import sys
print("script name is", sys.argv[0])
if len(sys.argv) > 1:
    print("there are", len(sys.argv)-1, "
                                arguments:")
    for arg in sys.argv[1:]:
        print(arg)
else:
    print("No arguments!")
```

os

- Some functions and variables in module `os`:
  - `environ`: A mapping object representing the string environment
  - `system(command)`: Execute the command (a string) in a subshell
  - `sep`: The character used by the operating system to separate pathname components
  - `pathsep`: The character conventionally used by the operating system to separate search path components (as in PATH)
  - `linesep`: The string used to separate (or, rather, terminate) lines on the current platform
  - `urandom(size)`: Return a string of size random bytes suitable for cryptographic use

More detail: ▸ os for 3.6.2rc2

Example

```
>>> import os
>>> os.environ['HOME']
'/Users/jerry'
>>> os.sep
'/'
>>> os.pathsep
':'
>>> os.linesep
'\n'
>>> os.system('pwd')
/Users/jerry/Desktop
0
>>> os.urandom(10)
b'\x86[&T\xc6\x8b\xbdN\xb91'
```