Basic Python Programming

Yilin Zheng

Dept. of Computer Science and Engineering

July 11, 2017

Outline I

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

Exceptions

- Errors detected during execution are called **exceptions** and are **not unconditionally fatal**
- Some common exceptions:
  - Indentation error
  - Type error
  - Name error
  - Index error
  - Attribute error
  - Key error
  - Value error
  - Overflow error
  - Zero division error

try...except...

- Use `try...except...` to handle exceptions
- Put statements within **try-block** while put error handlers within **except-block**

raise

- Use `raise` to raise an exception
- The error or exception that we can raise should be a **subclass** derived from the **Exception** class

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

with...as...

- To do other clean-up actions, use `with...as...`
- Form: `with EXPRESSION as VARIABLE:`

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

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

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Functional programming

- **Functional programming** is a **programming paradigm**, which treats computation as the **evaluation of mathematical functions** and **avoids changing-state and mutable data**
- There are some features of FP:
  - **First-class functions**: functions are **princeps**, they have equal status with other types of data, can be assigned to a variable, passed to or returned by other functions
  - **Pure functions**: Use **only expressions(functions)**, which has **no side effect**
  - **Type systems**: Use typed **lambda** calculus
  - **Referential transparency**: **No assignment statements**, so the value of a variable never changes once defined
- FP attaches importance to the **relations** between data, the function is **mathematical meaning**

Advantages

- Some advantages of FP:
  - Code can be more concise, programming can be faster
  - Closer to natural language, more comprehensible
  - Easier to debug and test
  - Friendly to implement concurrent programming

FP in Python

- Python is not a functional programming language, but it is a multi-paradigm language that makes functional programming easy to perform, and easy to mix with other programming styles
- We will learn three common functions in Python to do FP:
  - `lambda`
  - `map`
  - `reduce`
  - `filter`

lambda

- `lambda` is used to define **anonymous functions**
- Different from common functions which use `def` to define, we use `lambda` like in this form:
  `name = lambda parameters :operations`
- Call the name to use the function we defined

```
>>> add = lambda a,b : a+b #x, y are
                           paremeters, x+y is
                           operations
>>> add(1,2)
3
>>> power = lambda x,y : x**y
>>> power(2,3)
8
```

map

- `map` can **apply functions to every item of iterable** and **return a list of the results**
- Form: `map(function, iterable, ...)`, that is

```
map(f, iterable) equals to [f(x) for x in
                            iterable]
```

```
>>> list_1 = ['Minions', 'Spider-Man', '
                           Transformers']
>>> list(map(len, list_1))#the length of
                           strings
[7, 10, 12]
```

reduce

- `reduce` can apply a **rolling computation to sequential pairs of values in a list** and return the result
- In Python3, `reduce` has been moved into module `functools`, to use it, we need import it from this module
- Form: `reduce(function, iterable[, initializer])`

```
>>> list_2 = range(1, 101)#generate a list [1,
                          2,3,...,100]
>>> reduce((lambda x, y: x + y), list_2)#
                          calculate the sum
5050
```

- In this example above, `reduce` will calculate: 0+**1**=1, 1+**2**=3, 3+**3**=6, 6+**4**=10,..., 4950+**100**=5050

reduce

- `reduce` can accept a third parameter as the base of the calculation

```
>>> list_2 = range(1, 101)#generate a list [1,
                          2,3,...,100]
>>> reduce((lambda x, y: x + y), list_2, 8)#a
                          base value 8
5058
```

- After adding a third parameter, the calculation changes to be: **8**+**1**=9, 9+**2**=11,..., 4958+**100**=5058

filter

- `filter` can apply a **filtering rule on a list** and **returns a subset of that list after which the filtering rule is true**
- Form: `filter(function, iterable)`

```
>>> list_3 = range(1,10)#list from 1 to 9
>>> list(filter((lambda x : x % 2 == 0),
                                list_3))
[2, 4, 6, 8] #numbers which is even
>>> list(filter((lambda x : x % 3 == 0 and x %
                                 5 == 0), range(1,
                                80)))
[15, 30, 45, 60, 75] #numbers can be divided
                                by 3 and 5
```